An Autonomic Cross-Platform Operating Environment for On-Demand Internet Computing

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Vom Fachbereich Mathematik und Informatik der

Philipps Universität Marburg

angenommene Dissertation von

Stefan Paal

aus Germersheim, Deutschland

Marburg/Lahn, April 2010

Vom Fachbereich Mathematik und Informatik der
Philipps-Universität Marburg als Dissertation
am 28.04.2010 angenommen.

Erstgutachter: Prof. Dr.-Ing. Bernd Freisleben, Philipps-Universität Marburg

Zweitgutachter: Prof. em. Dr.-Ing. Reiner Kammüller, Universität Siegen

Tag der mündlichen Prüfung am 25.06.2010

Für meine Eltern Nada und Georg Paal

ACKNOWLEDGEMENTS

The accomplishment of this dissertation has taken me a long time and it looked not only to me like a never-ending project. There are many reasons for this; however, the most important one is that I spent much more time on improving the software implementations than on writing the documentation of the contributions. Over all this time, many people have continually pushed me through the work and I wish I could thank all of them personally for having been there and their unique support. Among them, there are several individuals, to whom I want to explicitly express my appreciation and gratitude.

First of all, I want to gratefully acknowledge my supervisor, Prof. Dr. Bernd Freisleben from the University of Marburg, for his long-standing support of my academic work. In the same line, I want to express my gratitude to Prof. em. Dr. Reiner Kammüller from the University of Siegen for the extensive and fruitful discussions about many research issues. Without their most valuable guidance and feedback, the present dissertation would have been not possible and many of my papers would miss the academic acknowledgments they have received in many conference talks.

I also wish to thank my colleagues and students at the University of Siegen, in particular Ralf Hofmann, Gerd Müller, Martin Hammel, Klaus Rühl and Kerstin Schmidt, who helped me through the beginning of the work on my dissertation. Their support for developing, implementing and testing the ODIX approach was essential for the overall accomplishment.

Next, I want to express my appreciation to the colleagues from the group MARS at the Fraunhofer Institute for Media Communication (IMK), especially Monika Fleischmann, Wolfgang Strauss, Dr. Jasminko Novak, Predrag Peranovic and Daniel Pfuhl. They particularly supported me the time I wrote many of my papers and presented them at various scientific conferences.

Furthermore, my special thanks go to Dr. Joachim Köhler, Dr. Stefan Eickeler, Ulrich Wernecke and Britta Schwipper from the group NetMedia at the Fraunhofer Institute for Intelligent Analysis and Information Systems (IAIS). Without their outstanding personal support and confidence, I would not have got the final push and the time for accomplishing the writing.

Last but certainly not least, I have to pay tribute to my family and friends, in particular to my parents Nada and Georg Paal, who believed in me all the time. They always gave me the backup I needed in difficult phases and I can never make up the time they all have spent on waiting for the completion of my dissertation.

ABSTRACT

The Internet has evolved into a global and ubiquitous communication medium interconnecting powerful application servers, diverse desktop computers and mobile notebooks. Along with recent developments in computer technology, such as the convergence of computing and communication devices, the way how people use computers and the Internet has changed people's working habits and has led to new application scenarios.

On the one hand, *pervasive computing*, *ubiquitous computing* and *nomadic computing* become more and more important since different computing devices like PDAs and notebooks may be used concurrently and alternately, e.g. while the user is on the move. On the other hand, the ubiquitous availability and pervasive interconnection of computing systems have fostered various trends towards the dynamic utilization and spontaneous collaboration of available remote computing resources, which are addressed by approaches like *utility computing*, *grid computing*, *cloud computing* and *public computing*.

From a general point of view, the common objective of this development is the use of *Internet applications on demand*, i.e. applications that are not installed in advance by a platform administrator but are dynamically deployed and run as they are requested by the application user. The heterogeneous and unmanaged nature of the Internet represents a major challenge for the on demand use of custom Internet applications across heterogeneous hardware platforms, operating systems and network environments. Promising remedies are autonomic computing systems that are supposed to maintain themselves without particular user or application intervention.

In this thesis, an *Autonomic Cross-Platform Operating Environment (ACOE)* is presented that supports *On Demand Internet Computing (ODIC)*, such as dynamic application composition and ad hoc execution migration. The approach is based on an integration middleware called *crossware* that does not replace existing middleware but operates as a self-managing mediator between diverse application requirements and heterogeneous platform configurations. A Java implementation of the *Crossware Development Kit (XDK)* is presented, followed by the description of the *On Demand Internet Computing System (ODIX)*.

The feasibility of the approach is shown by the implementation of an *Internet Application Workbench*, an *Internet Application Factory* and an *Internet Peer Federation*. They illustrate the use of ODIX to support local, remote and distributed ODIC, respectively. Finally, the suitability of the approach is discussed with respect to the support of ODIC.

ZUSAMMENFASSUNG

Das Internet hat sich zu einem allgegenwärtigen Kommunikationsmedium entwickelt, welches leistungsfähige Anwendungsserver, Desktop-Computersysteme und Notebooks gleichermaßen verbindet. Parallel zu den jüngsten Entwicklungen der Computertechnik und der damit verbundenen Annäherung von Computern und Kommunikationsgeräten hat das Internet die Arbeitsgewohnheiten der Nutzer verändert und zu neuen Anwendungsszenarien geführt.

Einerseits werden *Pervasive Computing*, *Ubiquitous Computing* und *Nomadic Computing* immer wichtiger, da unterschiedliche Geräte wie PDAs und Notebooks gleichzeitig und abwechselnd genutzt werden, z.B. während der Anwender sich von einem Ort zum anderen bewegt. Andererseits haben sich aus der allgegenwärtigen Verfügbarkeit und der fortschreitenden Vernetzung von Computersystemen neue Ideen zur dynamischen Nutzung und spontanen Kollaboration von entfernten Rechnersystemen entwickelt, die vor allem in *Utility Computing*, *Grid Computing*, *Cloud Computing* und *Public Computing* vorangetrieben werden.

Von einem allgemeinen Standpunkt betrachtet ist das gemeinsame Ziel die Nutzung von *Internet-Anwendungen auf Bedarf*, d.h. Anwendungen, welche nicht durch einen Administrator vorab installiert werden, sondern in dem Moment der Anfrage durch den Nutzer dynamisch verteilt und gestartet werden. Der heterogene Charakter und der dezentrale Aufbau des Internets bilden die hauptsächlichen Herausforderungen für den bedarfsgesteuerten Betrieb von kundenspezifischen Anwendungen über verschiedenartige Hardware-Plattformen, Betriebssystemen und Netzwerkumgebungen. Eine vielversprechende Lösung stellen hierzu autonome Rechnersysteme dar, die sich per Definition ohne besondere Nutzer- und Anwendungssteuerung selbst verwalten.

In dieser Arbeit wird eine Betriebsumgebung *Autonomic Cross-Platform Operating Environment (ACOE)* vorgestellt, die *On Demand Internet Computing (ODIC)* unterstützt. Der Ansatz basiert auf einer Integration Middleware namens *Crossware*, welche als Mittler zwischen verschiedenartigen Anwendungsanforderungen und heterogenen Plattformkonfigurationen dient. Weiterhin werden eine Java-Implementierung des *Crossware Development Kit (XDK),* gefolgt von einer Realisierung des *On-Demand Internet Computing Systems (ODIX)* beschrieben.

Die praktische Anwendung des Ansatzes wird anhand der Implementierungen einer *Internet Application Workbench*, einer *Internet Application Factory* und einer *Internet Peer Federation* veranschaulicht. Sie zeigen den Einsatz von ODIX für die Unterstützung in lokalen, entfernten und verteilten ODIC Anwendungsszenarien. Abschließend wird die Eignung des Ansatzes und der Implementierung mit besonderen Augenmerk auf die Unterstützung von ODIC diskutiert.

TABLE OF CONTENTS

TABLE OF FIGURES

ACRONYMS

ACL	Agent Communication Language
ACOE	Autonomic Cross-Platform Operating Environment
AJAX	Asynchronous JavaScript and XML
AOP	Aspect-Oriented Programming
API	Application Programming Interface
APT	Advanced Packing Tool
ASP	Application Service Provider
BOINC	Berkeley Open Infrastructure for Network Computing
BPEL	Business Process Execution Language
CaaS	Component-as-a-Service
CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
DLL	Dynamic Link Library
DNS	Domain Name System
DPE	Distributed Processing Environment
EAR	Enterprise Archive
ECMA	European Computer Manufacturers Association
EJB	Enterprise Java Beans
ESB	Enterprise Service Bus
FQCN	Fully Qualified Class Name
FTP	File Transfer Protocol
GNU	General Public License
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hyper Text Transport Protocol
IaaS	Infrastructure-as-a-Service
IIOP	Internet Inter-ORB Protocol
JAR	Java Archive
JCC	Java Class Collection
JCS	Java Class Space
JDBC	Java Database Connectivity
JDK	Java Development Kit
JEU	Java Execution Unit
JIT	Just In Time

JLM	Java Loadable Module
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JNLP	Java Network Launch Protocol
JRE	Java Runtime Environment
JRMP	Java Remote Method Protocol
JVM	Java Virtual Machine
JWS	Java Web Start
JXTA	Juxtapose, Language- and Platform-Independent Protocol for P2P Networking
LDAP	Lightweight Directory Access Protocol
ODAE	On-Demand Application Engine
ODC	On-Demand Computing
ODIC	On-Demand Internet Computing
ODIX	On-Demand Internet Computing System
OGSA	Open Grid Service Architecture
OSGI	Open Service Gateway Interface
P2P	Peer-to-Peer
PaaS	Platform-as-a-Service
PDA	Personal Digital Assistant
POJO	Plain Old Java Object
PVM	Process Virtual Machine
RAP	Rich AJAX Platform
RCF	Rich Client Framework
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
RPC	Remote Procedure Call
RPM	Red Hat Package Manager
SaaS	Software-as-a-Service
SMTP	Simple Message Transfer Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSH	Secure Shell
SVM	System Virtual Machine
UDDI	Universal Description, Discovery and Integration
USB	Universal Serial Bus

VM	Virtual Machine
WAR	Web Archive
WSDL	Web Service Description Language
XaaS	Everything-as-a-Service
XAML	Extensible Application Markup Language
XAR	Crossware Archive
XDK	Crossware Development Kit
XML	Extensible Markup Language
ZIP	Zip Drive File Format

1. Introduction

The introduction outlines the transition of the Internet from a dedicated network environment into a ubiquitous computing environment. The upcoming challenges along the evolution of *On Demand Internet Computing (ODIC)* are considered and the goal and subjects of this thesis are introduced. Then, the contributions of this thesis towards an *Autonomic Cross-Platform Operating Environment (ACOE)* are presented, which are followed by an overview of the rest of the work.

1.1 Motivation

1.1.1 Background

The Internet has evolved from a global communication medium towards a ubiquitous computing environment targeting the vision of Mark Weiser in which *"computers are available throughout the physical environment but effectively invisible to the user"* [379, 380]. Along with this evolution, the perception and use of Internet computing devices have changed, as illustrated below.

Client Computing. Originally, standard Internet applications, such as web browsers and email readers, were installed on a computing device and used to access remote Internet resources via common service protocols. The fixed feature set propelled the development and adoption of Internet applications for virtually all types of computing devices. In particular, the web browser turned into the standard interface to access remote information and applications provided in the Internet. Various advancements, such as Flash animations, Java applets and AJAX, helped to blur further the differences between desktop and browser applications. With the advent of custom browser plugins, such as Adobe AIR [2], Microsoft Silverlight [238] and Sun JavaFX [341], the scenario changed [216]. The Internet browser does no longer represent the main web application itself but it acts as a web application launcher for running *Rich Internet Applications* [224]. It enables users to run applications on-demand while he or she visits distinct web pages [295]. This trend even continued with the complete decoupling of the Internet application from the browser that is no longer needed to host the Internet applications. Related solutions, such as Sun Java Web Start [383] and Eclipse RCP [96], have fostered the trend towards *Rich Client Platforms* where application software is deployed via the Internet and installed on user request. The related software components are managed by using custom component repositories, such as Sun Java Store [346], to which developers might upload new components. Typically, the rich client platform periodically checks various repositories for new software releases and updates the local application installation; hence turning the Internet into a large-scale software deployment environment for client computing.

Nomadic Computing. At the same time, the convergence of computing and communication devices has changed people's working habits and has led to new applications, such as *nomadic and mobile computing* [196, 198]. Small computing devices with wireless link capabilities, such as PDAs and netbooks, enable people to connect to the Internet and access information while they are on the move [199, 251, 359], e.g. for querying a tourist information system by using a mobile phone [291]. The possibility of alternately using distinct computing devices, e.g. with different screen resolution and input interfaces, *"motivates the break from the traditional model of computation to a ubiquitous model that makes the user's entire environment available wherever it is required"* [82]. Particular software solutions, such as Sun J2ME [167], facilitate the instant execution of the same application on different mobile devices. In addition, roaming user profiles have been introduced to customize and provide the illusion of a pervasive application environment to the user across distinct computing devices and in different scenarios [197, 314]. Since computing terminals have become ubiquitously available, nomadic computing approaches have also gained attraction when fixed computing devices are used [206, 395], e.g. public information kiosks and desktop computers found in Internet cafes [221] or smart homes [98]. Besides launching a new application instance, a user may also request the migration of remote processes [82], e.g. relocating a terminal session from the office computer to the currently used notebook.

Service Computing. From another point of view, the provisioning and utilization of remote resources has also changed. Similar to the development on the client-side, server-side scenarios shifted from legacy web appliances towards custom services as in *service computing*. Web services based on standardized application protocols, such as SOAP, popularized the service-oriented computing model [170, 286] and has been early adopted by various manufacturers [366]. Application servers supporting standard programming models, such as Sun Enterprise Java Beans (EJB) [336], enable the cross-deployment of services independent of the actual vendor and have boosted the propagation of the web service model [366]. An Internet server is no longer only used to serve web pages but has evolved into an application server that is capable of running multiple service instances at the same time, e.g. an online shop and a help desk. Various efforts have also been made to benefit from the Internet growth [304] and related interconnection of large numbers of application servers, e.g. by combining the distributed computing power of server farms to surpass the limited performance progression of standalone computing systems [240], such as in Grid computing. In cloud computing, the service computing idea has been extended to the provisioning and utilization of common computing resources on user request, e.g. *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)* and *Software-as-a-Service (SaaS)* [220].

1.1.2 Evolution

As the progress in microelectronics technology goes on, computing resources become more and more a commodity good that are ubiquitously available [230]. In former times, specific applications and dedicated computer systems have been purchased and set up to match a given task setting exclusively. Nowadays, a user is able to switch applications and share computer systems to perform the same task in different use cases. In this scenario, a user typically does no longer want to be concerned with the computing details to perform a given task, e.g. how to retrieve a suitable application plugin for reading a document. Instead, the user wants to focus on the current task and expects the involved computing systems to provide the required resources on-demand, e.g. downloading code libraries as needed or adjusting the application configuration to match the current network proxy settings. Therefore and in contrast to traditional computing, the resources are no longer provided in advance and in a static manner but are allocated the time they are requested. This evolution towards *On-Demand Computing (ODC)* can be characterized as the shift from *resource-centric towards task-centric computation*, as shown in Figure 1.1.

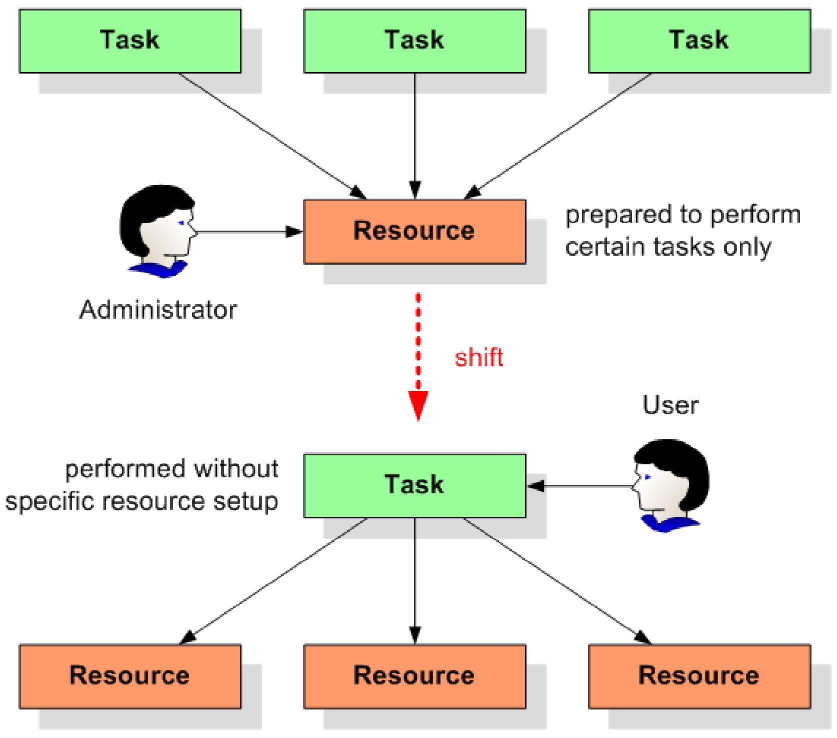

Task
Task
Task
Resource
prepared to perform certain tasks only
Administrator
shift
User
performed without specific resource setup
Task
Resource
Resource
Resource

Figure 1.1: Shift from Resource-Centric to Task-Centric Computing

In *resource-centric computing approaches*, an administrator prepares a given computing resource to perform certain tasks only and is characterized by a fixed assignment of resources. This procedure is well-known in administered environments like for example in enterprise networks with dedicated servers, such as database and mail servers. In *task-centric computing approaches*, there is no fixed assignment and a user can ideally process a task on various computing resources. They have been not set up for a specific computing purpose but are spontaneously adjusted to perform the present task. This idea has become very common when requesting computing resources in changing task scenarios like in nomadic computing, Grid computing and public computing. With the hype around cloud computing [150] and its implementation by commercial providers, e.g. Amazon EC2 and Google AppEngine [215], the idea of ODC has recently gained much support and attraction.

1.1.3 Vision

A particular vision in the evolution of ODC is to break up the resource-centric relationship among user, application and computing device and to replace it with a task-centric assignment. In place of static task configuration, dynamic mediation of computing resources enables on-demand task processing without user intervention. A specific feature of ODC is the on-demand provision of suitable applications to process the submitted user tasks on unsupervised computing resources, as illustrated in Figure 1.2.

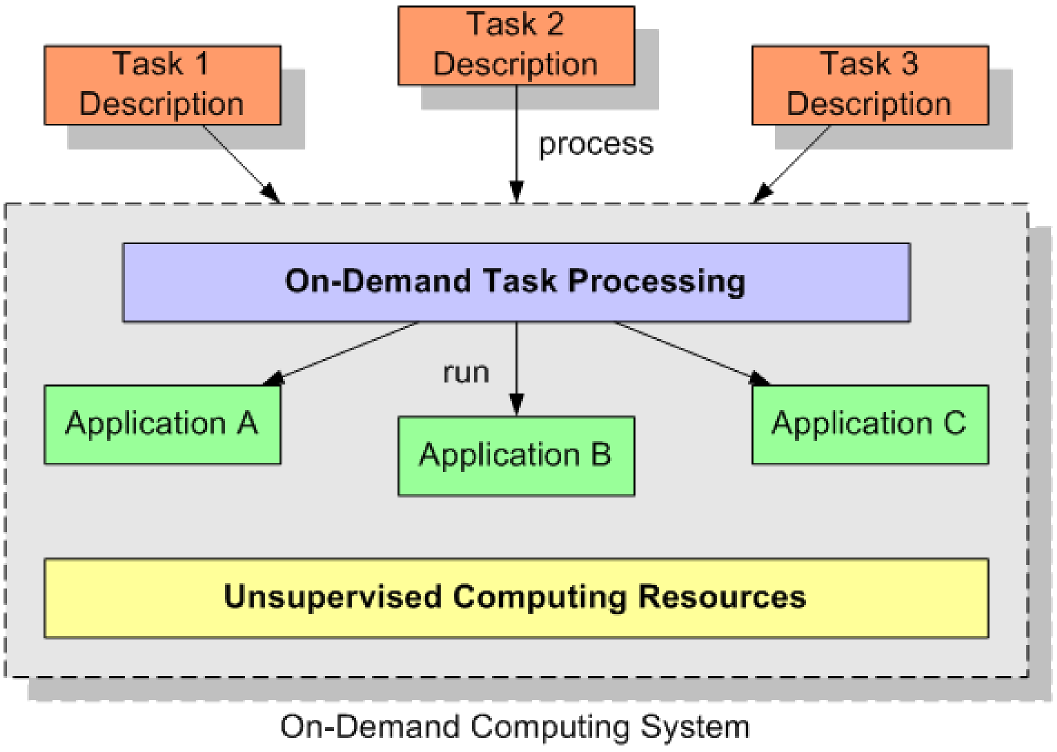


Figure 1.2: On-Demand Computing System

The task descriptions do not contain specifications about which application is to be launched and which computing resources may be allocated to process the task. The *On-Demand Computing System* is free to set up a matching processing configuration, e.g. following a given computing policy to minimize the resource costs or reduce the overall completion time.

1.2 Focus

1.2.1 Goal

The goal of this thesis is to apply the vision of ODC [51, 110] on the Internet and to elaborate an approach for *On-Demand Internet Computing (ODIC)*. In particular, integrated application systems to enable *local task processing*, *remote task processing* and *distributed task processing* on-demand are to be implemented. A prerequisite is to enable the dynamic provision and configuration of networked resources to run Internet applications on heterogeneous computing systems by user request, as illustrated in Figure 1.3.

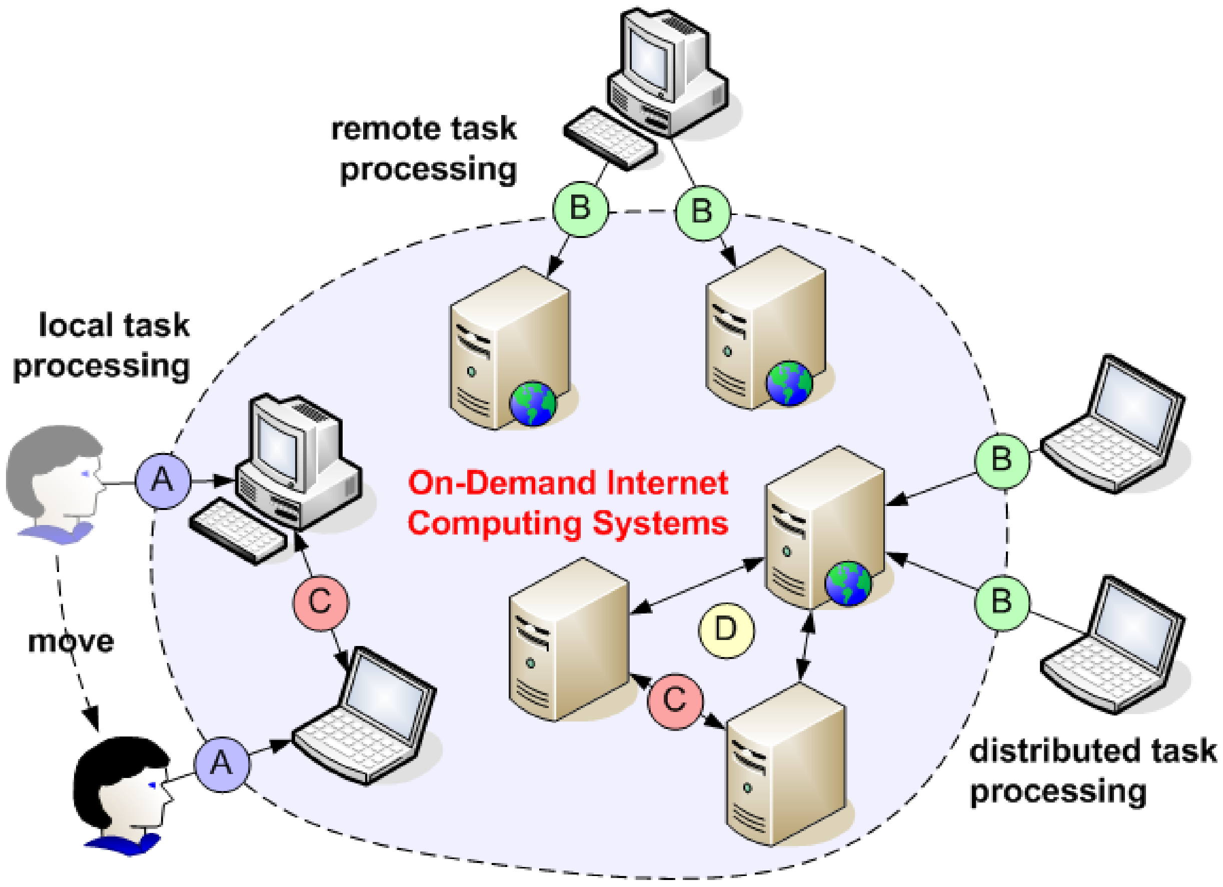

remote task processing
local task processing
On-Demand Internet Computing Systems
distributed task processing
move
A
A
B
B
B
B
C
C
D

Figure 1.3: Running On-Demand Internet Applications on Networked Resources

Internet applications do not have to be installed in advance but are deployed, composed and launched on the fly. Nomadic users may switch among heterogeneous computing devices and access their personal settings, applications and documents while being on the move (A). Remote computing resources can be assigned to instantaneously run custom applications without administrator intervention (B). Running application components may migrate from one application server to another and seamlessly continue their computation (C). Finally, multiple application servers may dynamically grouped in federations to balance time-consuming task processing requests (D).

1.2.2 Challenges

The realization of *On-Demand Internet Computing (ODIC)* encounters particular challenges that reflect the spontaneous use of computer devices to run custom applications in unmanaged and heterogeneous environments [98]. The major issue is the unpredictable constellation of application requirements and platform capabilities which makes it difficult to provide an all-purpose application configuration or ensure a uniform platform administration, as shown in Figure 1.4.

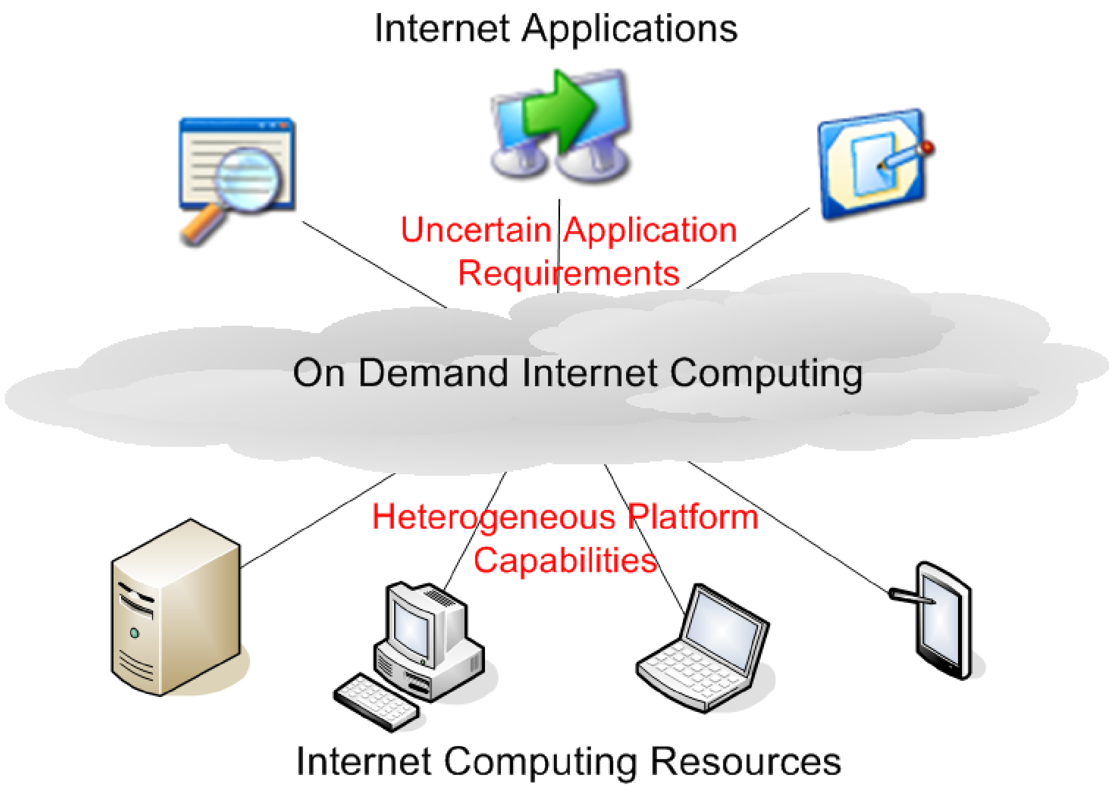

Internet Applications
Uncertain Application Requirements
On Demand Internet Computing
Heterogeneous Platform Capabilities
Internet Computing Resources

Figure 1.4: Uncertain Application Requirements and Heterogeneous Platform Capabilities

Required resources cannot be provided and allocated in advance but have to be determined and requested in the moment they are needed. In turn, resources are alternately utilized in different application scenarios and the relationship of device user, application installation and employed computer device is untied and replaced by a dynamic assignment. Application components may be spread and linked across various Internet nodes and moved while they are in use. Due to its large extent and diversity, the Internet also renders precautionary configuration attempts of application deployers and platform administrators infeasible. This results in Internet application systems that cannot be actually prepared by administered approaches to support yet unknown applications. A suitable approach is to manage dynamic and unspecified scenarios without manual user intervention. Promising remedies are *autonomic computing systems* that are supposed to maintain and adjust themselves according to the current application scenario.

1.2.3 Subject

The subject of this thesis is the elaboration, design and implementation of an *Autonomic Cross-Platform Operating Environment (ACOE)* supporting ODIC. It is supposed to enable the seamless and dynamic employment of Internet applications across different operating systems and platforms by introducing a *self-managing integration middleware* called *crossware*, as shown in Figure 1.5.

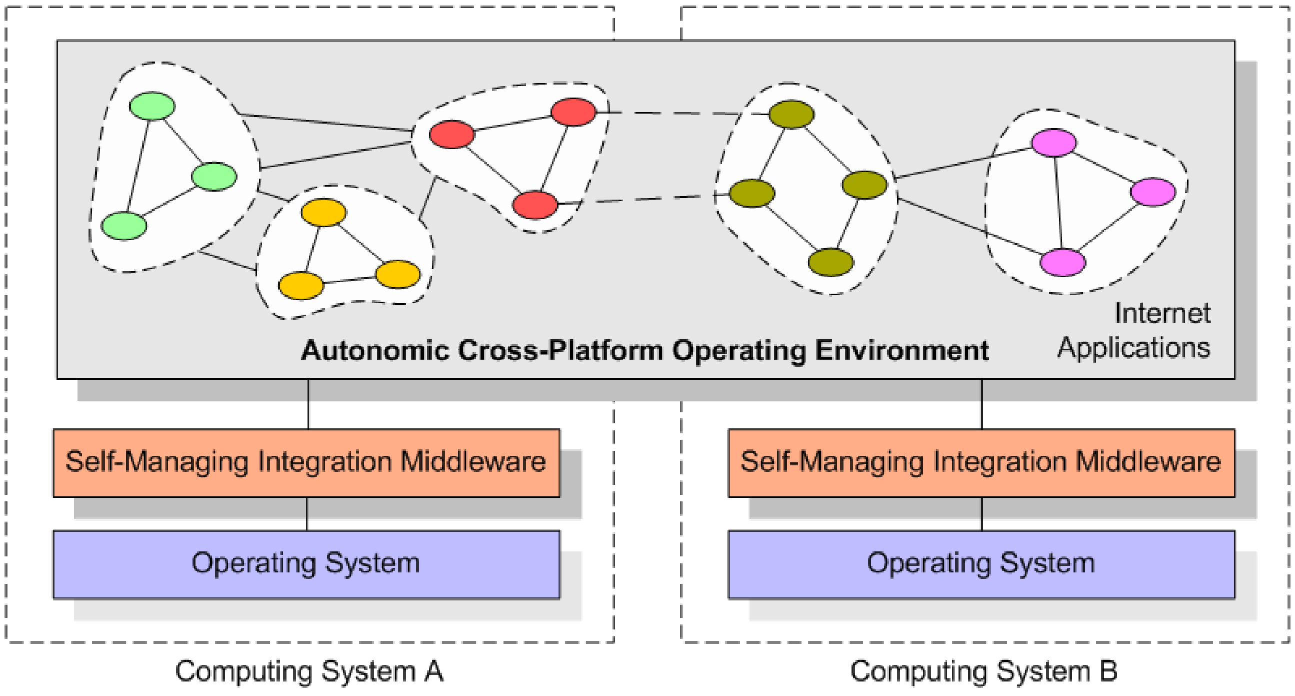

Internet Applications
Autonomic Cross-Platform Operating Environment
Self-Managing Integration Middleware
Operating System
Self-Managing Integration Middleware
Operating System
Computing System A
Computing System B

Figure 1.5: Autonomic Cross-Platform Operating Environment

The self-managing integration middleware is manually installed on participating computing systems in advance and appropriately configured to exploit the specific characteristics of the underlying operating system and hardware platform. It is not supposed to replace conventional middleware approaches but acts as a mediator to uniformly interact with heterogeneous platform installations, resources and features. Inspired by the autonomic computing approach of IBM [250], it particularly supports the self-managed deployment, composition, hosting, customization, interconnection and migration of Internet applications without manual user intervention. The resulting ACOE virtually hides the use of distinct computing systems and provides the illusion of a pervasive application environment to the user and the Internet application. In particular, the users, the developers and the applications are relieved to deal with the current platform configuration.

1.3 Overview

1.3.1 Contributions

The major contributions of the thesis are: elaborating the challenges of *On Demand Internet Computing (ODIC)*, introducing the features of an *Autonomic Cross-Platform Operating Environment (ACOE)*, and realizing and evaluating the *Crossware Development Kit (XDK)* and the *On Demand Internet Computing System (ODIX)*, as shown in Figure 1.6 and outlined below.

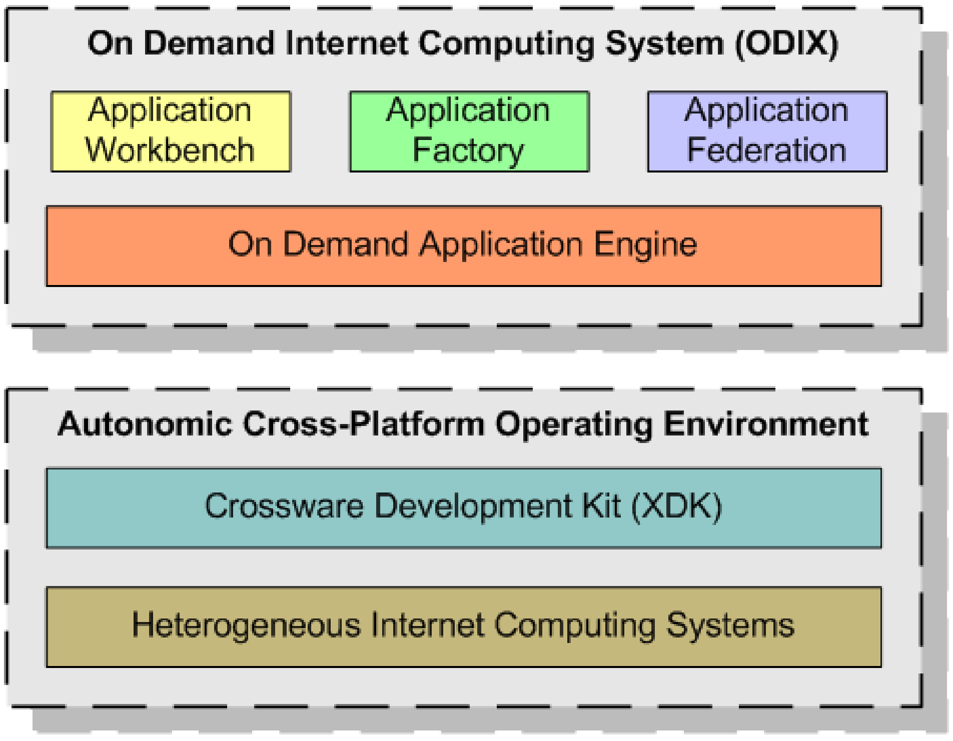

On Demand Internet Computing System (ODIX)
Application Workbench
Application Factory
Application Federation
On Demand Application Engine
Autonomic Cross-Platform Operating Environment
Crossware Development Kit (XDK)
Heterogeneous Internet Computing Systems

Figure 1.6: Contributions of the Thesis

- **On Demand Internet Computing (ODIC).** The first contribution is the elaboration of the scope and the vision of On Demand Internet Computing (ODIC). The Internet and its challenges for running Internet applications are presented. Application scenarios of traditional Internet computing are considered and related assets along with the involved user roles are identified. After presenting the original ideas of On Demand Computing (ODC), the proposed shift from resource-centric to task-centric computing and the replacement of static resource allocation by dynamic resource assignment are described. The related facets of ODIC and various visions supporting nomadic computing and utility computing in the Internet are outlined. A reflection relates the idea and needs of ODIC to existing Internet computing approaches like web computing and peer-to-peer computing.

- **Autonomic Cross-Platform Operating Environment (ACOE).** A new approach towards ODIC is presented based on an *autonomic cross-platform operating environment.* The goal is to deal with uncertain application scenarios in an Internet environment by separating the application configuration, the resource administration and the environment customization. To this end, the challenges imposed by the Internet are examined and the design of a cross-platform operating environment for dealing with related cross-platform issues is presented. The need for automation in an unmanaged environment like the Internet is illustrated and the ideas of autonomic computing are explored. The outcome is a self-managing infrastructure that replaces the manually performed tasks of the user roles in Internet computing scenarios into *self-managing operations.* Finally, a review of existing solutions motivates the need for a different implementation.

- **Crossware Development Kit (XDK).** The implementation of the autonomic cross-platform operating environment in Java is described. It pursues the automation of specific computing assets by separating the deployment, composition, hosting, customization, interconnection and migration of applications. To this end, a *self-managing integration middleware* is implemented that performs the virtualization and integration of platform resources as well as their automation. The outcome is the Crossware Development Kit (XDK) that represents a Java application framework that hides platform-specific details from the application while mediating between application requests and platform capabilities in a self-managing way. As a result, applications can be run and moved among peers without having been explicitly installed and configured on each node in advance.

- **On Demand Internet Computing System (ODIX).** The application of the XDK is demonstrated by means of the On Demand Internet Computing System (ODIX). An *On-Demand Application Engine* can be dynamically deployed via the Internet, supports the integration of multiple applications and interacts with peer instances to provide the illusion of a pervasive application environment across heterogeneous computing devices. The *Internet Application Workbench* can be used by nomadic users to seamlessly launch custom applications on alternating computing devices without the need of manual user profile synchronization. Another development is the *Internet Application Factory* that supports spontaneous deployment and remote execution of Internet applications on remote computing devices. The *Internet Application Federation* shows the execution migration of running Internet applications between networked computing devices.

1.3.2 Publications

Aspects of the work described in this thesis have been partially published in the following journals and conference proceedings:

P1. Paal, S. *ODIX: An On-Demand Internet Application Workbench*. Proceedings of the 9th International Conference on Internet Computing (ICOMP 2008). Las Vegas, USA. CSREA 2008. pp. 342-348.

P2. Paal, S., Bröcker, L., Borowski, M. *Supporting On-Demand Collaboration in Web-Based Communities.* Proceedings of the 17th IEEE International Conference on Database and Expert Systems Applications (DEXA 2006). Krakow, Poland. IEEE 2006. pp. 293-298.

P3. Paal, S., Kammüller, R., Freisleben, B. *Self-Managing Application Composition for Cross-Platform Operating Environments.* Proceedings of the 2nd IEEE International Conference on Autonomic and Autonomous Systems (ICAS 2006). Silicon Valley, USA. IEEE 2006. p. 37.

P4. Paal, S., Kammüller, R., Freisleben, B. *Crossware: Integration Middleware for Autonomic Cross-Platform Internet Application Environments.* Journal on Integrated Computer-Aided Engineering. Vol. 13, Nr. 1. IOS Press 2006. pp. 41-62.

P5. Paal, S., Kammüller, R., Freisleben, B. *Crosslets: Self-Managing Application Deployment in a Cross-Platform Operating Environment.* Proceedings of the 3rd International Conference on Component Deployment (CD 2005). LNCS 3798. Grenoble, France. Springer 2005. pp. 51-65

P6. Paal, S., Kammüller, R., Freisleben, B. *An Autonomic Cross-Platform Operating Environment for On Demand Internet Computing.* Demonstration on the 6th International Middleware Conference (MW 2005). Grenoble, France. 2005.

P7. Paal, S., Kammüller, R., Freisleben, B. *Application Object Isolation in Cross-Platform Operating Environments.* Proceedings of the 6th International Symposium on Distributed Objects and Applications (DOA 2005). LNCS 3761. Agia Napa, Cyprus. Springer 2005. pp. 1047-1064.

P8. Paal, S., Kammüller, R., Freisleben, B. *Dynamic Software Deployment with Distributed Application Repositories.* 14. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005). Informatik aktuell. Kaiserlautern, Germany. Springer 2005. pp. 41-52.

P9. Paal, S., Novak, J., Freisleben, B. *Kollektives Wissensmanagement in virtuellen Gemeinschaften.* Wissensprozesse in der Netzwerkgesellschaft. transcript Verlag 2004. pp. 119-143.

P10. Paal, S., Kammüller, R., Freisleben, B. *Supporting Nomadic Desktop Computing using an Internet Application Workbench.* Proceedings of the 5th International Conference and Workshop on Distributed Objects and Applications (DOA 2004). Larnaca, Cyprus. Springer 2004. pp. 40-43.

P11. Paal, S., Kammüller, R., Freisleben, B. *A Cross-Platform Application Environment for Nomadic Desktop Computing.* Proceedings of the International Conference on Objects, Components, Architectures, Services, and Applications for a NetworkedWorld (NODE 2004). LNCS 3263. Erfurt, Germany. Springer 2004. pp. 185-200.

P12. Paal, S., Kammüller, R., Freisleben, B. *Self-Managing Remote Object Interconnection.* Proceedings of the 15th International Conference and Workshop on Database and Expert Systems Applications (DEXA 2004). Zaragoza, Spain. IEEE 2004. pp. 758-763.

P13. Paal, S., Kammüller, R., Freisleben, B. *Separating the Concerns of Distributed Deployment and Dynamic Composition in Internet Application Systems.* Proceedings of the 4th International Conference on Distributed Objects and Applications (DOA 2003). LNCS 2888. Catania, Italy. Springer 2003. pp. 1292-1311.

P14. Paal, S., Kammüller, R., Freisleben, B. *Java Remote Object Binding with Method Streaming.* Proceedings of the 4th International Conference on Objects, Components, Architectures, Services and Applications for a Networked World (NODE 2003). Erfurt, Germany, 2003. pp. 230-244.

P15. Paal, S., Kammüller, R., Freisleben, B. *Java Class Deployment with Class Collections.* Objects, Components, Architectures, Services, and Applications for a NetworkedWorld. LNCS 2591. Erfurt, Germany. Springer 2003. pp. 135-151.

P16. Paal, S., Kammüller, R., Freisleben, B. *Customizable Deployment, Composition and Hosting of Distributed Java Applications.* Proceedings of the 3rd International Conference on Distributed Objects and Applications (DOA 2002). LNCS 2519. Irvine, USA. Springer 2002. pp. 845-865.

P17. Paal, S., Kammüller, R., Freisleben, B. *Java Class Deployment with Class Collections.* Proceedings of the 3rd International Conference on Objects, Components, Architectures, Services and Applications for a Networked World (NODE 2002). Erfurt, Germany. 2002. pp. 144-158.

P18. Paal, S., Kammüller, R., Freisleben, B. *Java Class Separation for Multi-Application Hosting.* Proceedings of the 3rd International Conference on Internet Computing (IC 2002). Las Vegas, USA. CSREA 2002. pp. 259-266.

P19. Paal, S., Kammüller, R., Freisleben, B. *Dynamic Composition of Web Server Functionality over the Internet.* Proceedings of the 6th International WebNet World Conference of the WWW, Internet, and Intranet (Webnet 2001). Orlando, USA. AACE 2001. pp. 967-972.

P20. Paal, S., Kammüller, R., Freisleben, B. *Distributed Extension of Internet Information Systems.* Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2001). Anaheim, USA. IASTED 2001. pp. 38-43.

1.3.3 Thesis Map

The remainder of the thesis is organized as follows. In Chapter 2, the facets and vision of On Demand Internet Computing (ODIC) are presented and related approaches are reviewed. In Chapter 3, an autonomic cross-platform operating environment is introduced that is especially designed to support ODIC. In Chapter 4, the corresponding implementation of the Crossware Development Kit (XDK) is illustrated and its features are compared with related work. In Chapter 5, the On Demand Internet Computing System (ODIX) that is implemented on top of the XDK is presented and its application for running Internet applications on-demand is demonstrated. Chapter 6 concludes this thesis and outlines areas for future work.

2. Towards On-Demand Internet Computing

2.1 Introduction

In this chapter, the motivation and scope of *On Demand Internet Computing (ODIC)* are determined and common objectives are deduced. The definition of the Internet is presented and its characteristics are outlined. Then, application scenarios of Internet computing are considered and related assets as well as involved user roles are deduced. Afterwards, the trend from resource-centric to task-centric computation as proposed in *On Demand Computing (ODC)* is highlighted. Transferring this approach to the Internet, the facets of *On Demand Internet Computing (ODIC)* are elaborated and several visions of use are presented. The chapter ends with a review of related Internet computing approaches and their support of ODIC.

2.2 Internet

2.2.1 Definition

To delimit the scope of Internet computing, a definition of the Internet environment is given and how it differs to other network installations. A good starting point is the comparison with Intranet and Extranet as shown in Figure 2.1.

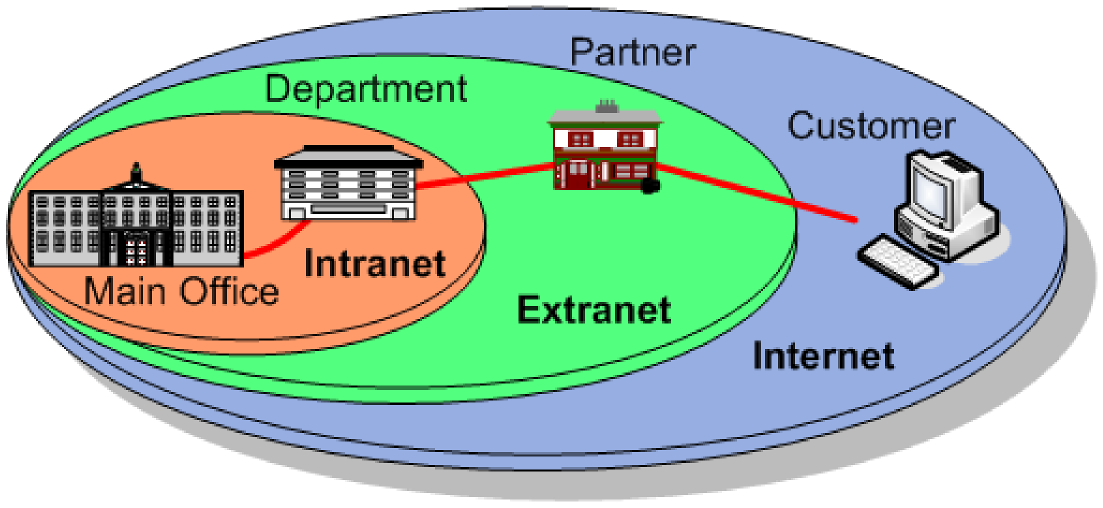


Figure 2.1: Intranet, Extranet and Internet

Intranet. An *Intranet* is a closed network, typically limited to a single organization unit, such as a department or enterprise. It is run by a single authority that manages the employed computing devices and network components. Though there is no need to rely on global standards, most Intranets employ well-known technologies. In addition, they introduce suitable profiles to ensure a homogeneous computing environment that can be easily managed, e.g. the same MS Windows operating system on every computer is installed and using a central user profile management.

Extranet. Due to business demands, there is a valuable need to access Intranet resources across multiple organizational units, such as found among project partners or within a joint enterprise. While the network components within an Intranet are actually still not accessible from outside, others are installed in the *Extranet* that may be accessed from everywhere, such as a public web server. They provide selected access to Intranet resources like a customer database and follow global communication standards, such as the network protocol HTTP.

Internet. In contrast to an Intranet and an Extranet, the *Internet* is not managed by a single authority, limited to a certain purpose or closed to an organizational unit. The Internet is observed as a global network communication medium that enables different computing devices to communicate. It is based on global and publicly accepted communication standards and services, such as the *Domain Name System (DNS)*, but does not impose the installation of a specific operating system or use of a particular hardware component.

2.2.2 Characteristics

The Internet poses several constraints upon the deployment and hosting of distributed applications [53]. A summary of the underlying characteristics is shown in Figure 2.2 and described below

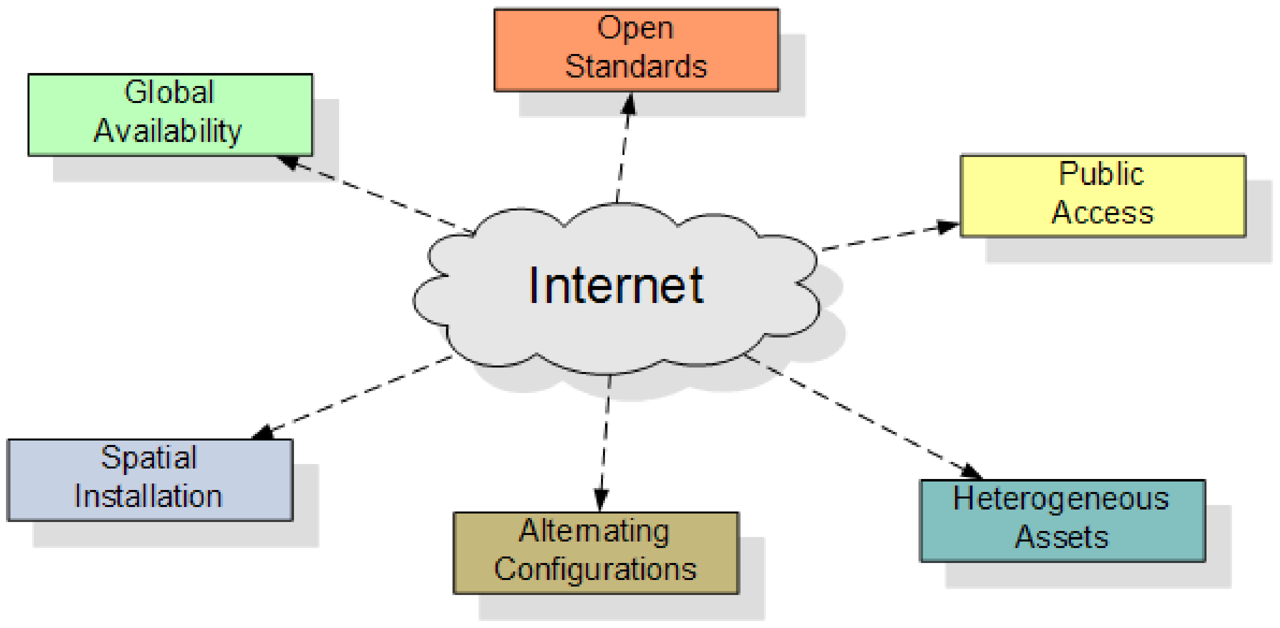


Figure 2.2: Characteristics of the Internet

Global Availability. Once an Internet application has been installed, it can be used from any place around the world. There is no need for switching a particular link between two Internet nodes but the Internet infrastructure is available all the time, e.g. proving 24/7h access to a news service. From this point of view, an Internet application may be concurrently used by a large number of users which may lead to performance and security problems.

Spatial Installation. The Internet is not limited to a certain location but represents a global communication network. Related applications are distributed among various Internet nodes that have been set up separately. The installation and configuration of an application cannot be performed for every node by a single authority. Local administrators and users do not follow a common policy but use different setup configurations, such as firewall rules and directory organization.

Open Standards. An important key stone for the success of the Internet is the foundation on open standards. The interoperability of Internet applications is not bound to a particular implementation but on public specifications. In this line, the open source idea gained much attraction by offering solutions that may be used instead of and mixed with commercial software, such as Apache web server or the Internet browser Mozilla Firefox.

Public Access. Another feature of the Internet is the provision of public access to any networked resource. Apart from the dial-in costs to the local Internet provider, there are no further charges, such as time or volume based connection fees. In conjunction with the global availability and use of open standards, this turned Internet computing into a ubiquitous approach that has influenced many areas of everyday life, e.g. offering the online encyclopedia Wikipedia at no charge.

Heterogeneous Resources. The Internet is a global conglomerate of highly diverse computing devices, operating systems and applications. In contrast to an Intranet, there is no common policy which ensures a homogeneous operating environment. Due to this, Internet computing has to deal with heterogeneous assets, e.g. by distributing a specific implementation for every kind of supported operating systems or by relying on virtual runtime environments, such as Sun Java.

Alternating Configurations. Another issue of the Internet is the diversity of platform installations that leads to alternating configuration scenarios from node to node. In effect, each Internet application is typically installed and configured separately without considering a concurrent installation or shareable resources, such as common code libraries. Moreover, it is difficult to dynamically deploy and configure applications across multiple nodes without particular user intervention.

2.2.3 Challenges

Considering the characteristics, the Internet represents a particular network environment that imposes various challenges for running Internet applications, as described below.

Concurrent Repositories. In contrast to an Intranet, the Internet is a public space where distributed and separately managed application repositories co-exist. They allow the introduction of particular features needed for large scale deployment scenarios, such as caching and fault tolerance. However, application components may be concurrently deployed to various application repositories and the availability of a component in well-known repositories cannot be guaranteed. Particular problems are the retrieval of software components from multiple remote repositories, the discovery of code repositories in unknown network scenarios and the encapsulation and transmission of related software packages. Furthermore, security issues, such as proving the authenticity and validity of code packages, must be tackled.

Software Evolution. Along with concurrent code repositories and software deployment, the problem of identifying and employing appropriate code packages on-demand shows up. In a typical scenario, an administrator identifies the right code libraries and configures the application accordingly. However, there is no common guideline for Internet environments how to tag and to distribute code packages with additional attributes apart from the name. Moreover, different code packages may be compatible and whereas the same code package could be provided in different variants, e.g. one with a command-line interface and another using a GUI dialog. Dynamic dependency resolution considering the current application scenario and available code packages is a further issue in this context.

Portable Systems. Similar to nomadic users, computing devices have turned into portable systems, such as notebooks, PDAs or smart phones. They have to deal with changing network configuration and interconnection scenarios, e.g. roaming through network cells. Another problem is the unpredictable change from online to offline operation. In this context, existing network links may have to be re-established and switched. Furthermore, Internet applications cannot rely on a permanent connection to a remote service but have to maintain its execution state and data in offline mode. This leads to another problem, namely resource consumption and capabilities. Small computing devices may only be used to run a limited set of applications and thus resource sharing across applications should be considered.

Runtime Environment. Another problem is caused by the heterogeneous nature of the Internet and the changing use of different resources. On the one hand, it is not possible to configure each resource on every node in the same way. On the other hand, an application cannot be individually implemented for each type of resource. Often used solutions are virtual runtime environments, such as provided by Sun Java. A developer is able to compile an application into the intermediate Java byte code that can be instantaneously run on any host with a Java Virtual Machine (JVM) installed. However, this approach does not address different host configurations that may affect the application composition, e.g. the lookup of required resources.

Large-Scale Extension. The large extent of the Internet and the comparatively limited bandwidth of shared connections represent a particular challenge for the distribution of application code and execution data. In contrast to a local network where bandwidths of up to 1 GBit/s are available, an Internet connection is less capable and may even fail for several seconds. Particular scenarios are mobile computing nodes, such as PDAs, which are usually equipped with low-bandwidth solutions, such as Bluetooth or Wi-Fi. Consequently, the amount of data that has to be transmitted from the source to the target host must be considered, e.g. by querying a local cache for already downloaded application code.

Insecure Infrastructure. The Internet was designed to bridge distinct computing networks and administration domains. On the one hand, this approach follows the basic idea to create a public networking infrastructure without imposing a one-for-all global network management; on the other hand it inherits security problems by incorporating unknown and potentially dangerous parties. Apart from intercepting and manipulating network communication, Internet computing systems have proven to be vulnerable against hacker intrusions, denial-of-service attacks and malicious code received from remote sites. In this scenario, on-demand computing has to address the integrity of code deployment, object communication and runtime environment.

Separated Administration. A basic difference of Intranets and Internet is the management of their resources. The Internet is not ruled by a single organization but independently managed by diverse authorities. There is no central instance that may be used to update software installations or synchronize user profiles across different nodes at the same time. Instead, each node has to manage itself according to the application scenario, user context and network configuration. A related problem is the lack of a common configuration policy. There is no guideline where to install application libraries or how to access system resources. This makes it difficult for a developer to prepare the operation of an application in a foreign computing environment and usually requires a separate manual installation process.

2.3 Internet Computing

In this section, selected application scenarios of Internet computing are compared to refine the picture. Afterwards, related assets and user roles are deduced to elaborate the tasks that have to be performed manually.

2.3.1 Application Scenarios

The Internet is used for various computing purposes and applications. With respect to the characteristics of the Internet presented in Section 2.2.2, several application scenarios to determine the basic assets of Internet computing are considered.

Personal Computing. Due to global availability and open standards, remote Internet resources may be accessed using distinct applications individually installed and configured on a personal computing device, as shown in Figure 2.3.

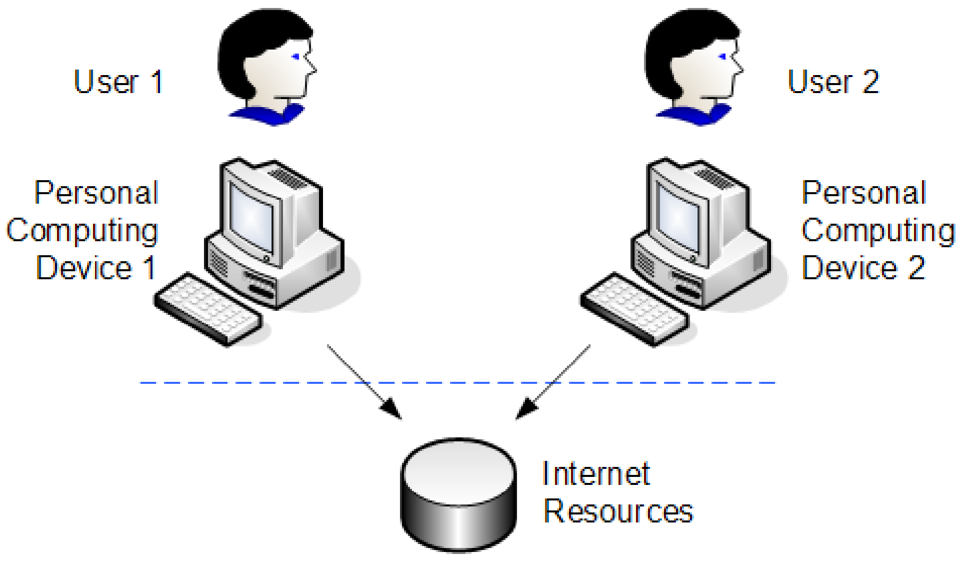


Figure 2.3: Exclusive Use of a Local Computing Device

As an example, different web browser implementations can be used to access web pages in the same way. The same is valid for other types of services, such as Telnet, FTP or SSH. In *personal computing* scenarios, the computing device is usually not shared with other users and the user does not employ other devices, such as PDAs or laptops. This results in a fixed assignment of device, application and user.

Pervasive Computing. Another use of the Internet is *pervasive computing* and the breakup of strict device assignment seen in personal computing by alternate and unconscious employment of various computing devices [251], as shown in Figure 2.4.

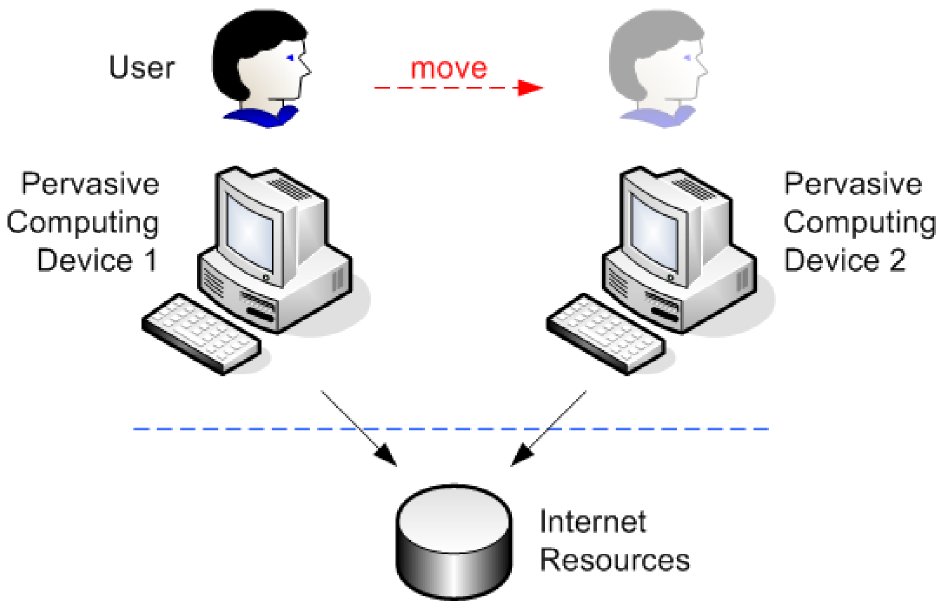


Figure 2.4: Alternate Employment of Distinct Computing Devices

After the user has finished using device 1, he or she may move from device 1 to device 2 and continue his or her work, e.g. while walking between distributed information desks in a museum or bulletin boards spread in a smart home environment. Since the user is not strictly bound to a single computing device and in turn the same device may be alternately used by different users, each device is dynamically customized for providing a personal working environment to the user.

Nomadic Computing. A further step along this line is the alternate and mobile use of portable computing devices, such as laptop and handheld computers, in conjunction with mobile communications technologies, as shown in Figure 2.5.

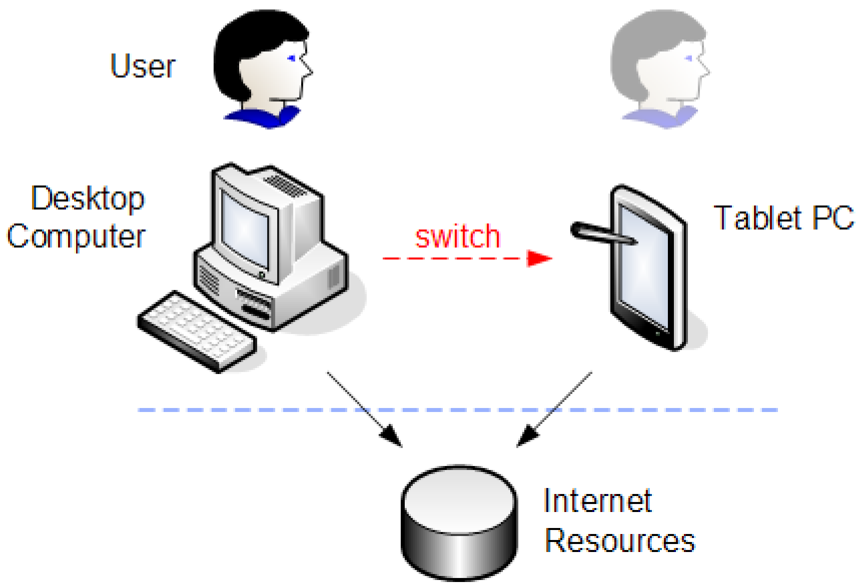


Figure 2.5: Switching to Another Type of Computing Device

In *nomadic computing*, users are enabled to access the Internet, programs and data at distinct locations. A user may take the computing device with him or engage different personal computing devices which, in contrast to pervasive computing, are typically assigned to the same user. Nomadic computing is designed to provide the system-level support for users who travel and switch computing devices while being on the move. The system support is meant to make this move as seamless and transparent as possible.

Remote Computing. The global availability of the Internet and use of open standards allow to access computing resources in a uniform way though running in a heterogeneous environment. In *remote computing*, a dedicated computer offers resources while other machines access these resources remotely via the Internet, as shown in Figure 2.6.

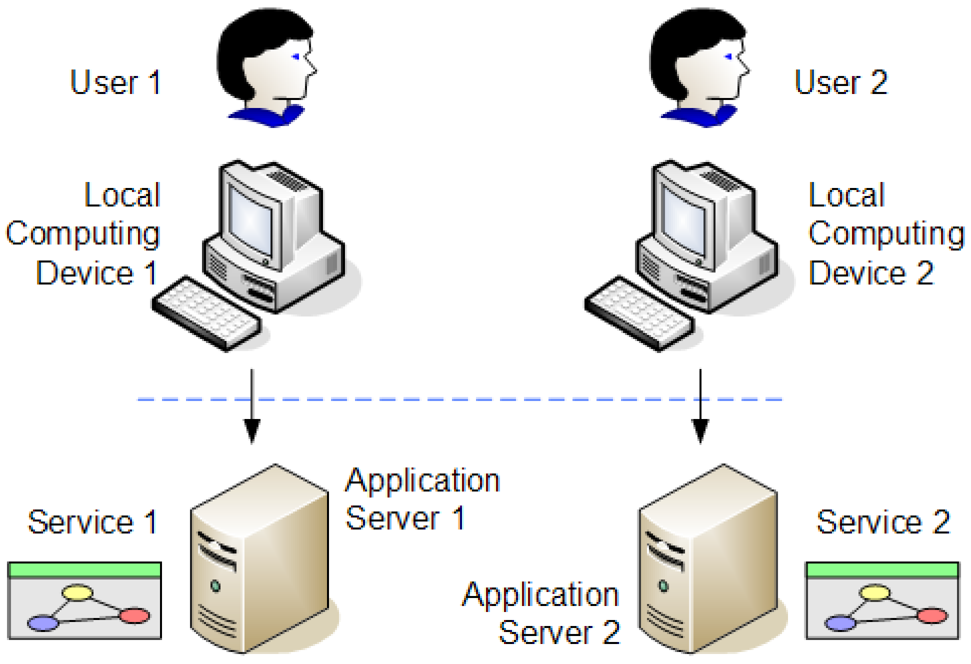


Figure 2.6: Exclusive Assignment of an Application Server

A popular example is the *World Wide Web (WWW)* that is based on the use of web standards, such HTML and HTTP. A web browser can interact with every web server without any modification. An advanced option is the installation of custom web services that are offered using WSDL, SOAP and UDDI [322]. For remote access on application objects like in the object-oriented approach, there are cross-platform solutions, e.g. CORBA.

Shared Computing. In another application scenario, the Internet supports the outsourcing of distinct computing services to *Application Service Providers (ASP)*, such as web hosting companies or computer centers. In *shared computing*, enterprises let third-parties operate and maintain their applications and IT infrastructure, as shown in Figure 2.7.

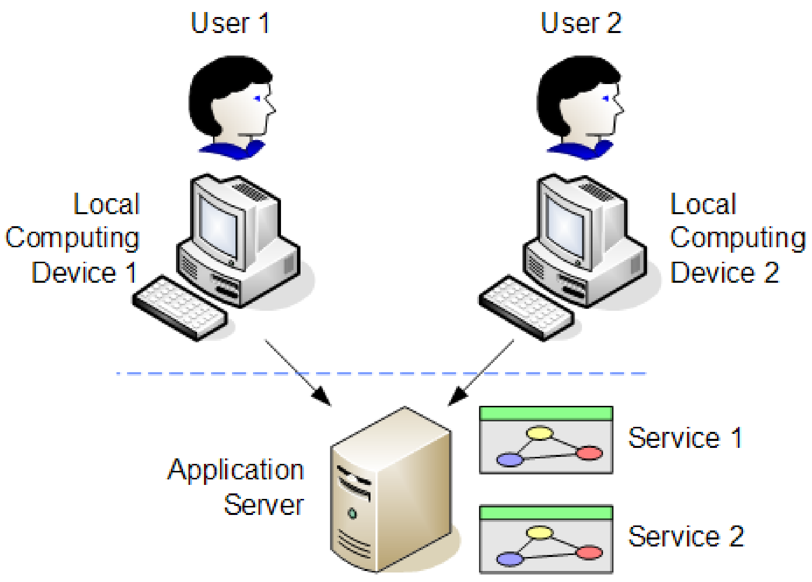


Figure 2.7: Sharing an Application Server with Multiple Services

Service providers seek to minimize the overall costs by untangling the fixed assignment of computing resources, applications and customers. Instead, a service provisioning model allows the dynamic allocation of computing resources as needed. Customers may then choose the best suitable application service provider for each task on a pay-per-use basis.

Parallel Computing. The focus of *parallel computing* in the Internet is the concurrent use of various computing devices to perform a computational task, as shown in Figure 2.8.

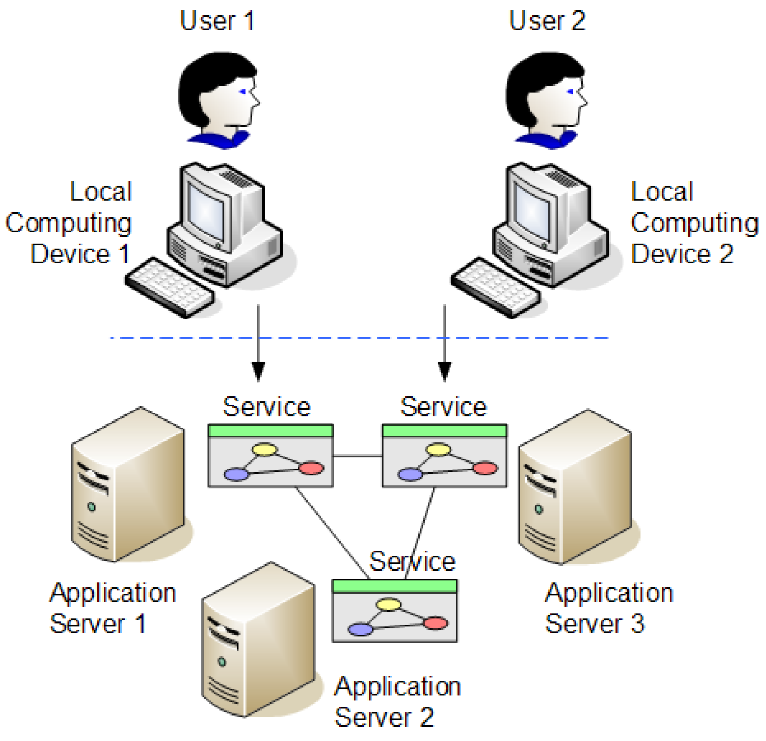


Figure 2.8: Concurrent Employment of Multiple Application Servers

A related approach is Grid computing that crosses administrative domains and different device configurations [116]. Common to all grids is their central management by a distinct grid node that splits a computational task into smaller pieces, deploys them on available nodes and assembles the partial results after completion. Typical is also the use of a common Grid installation based on open standards, such as *Globus* [135], which is deployed on each grid node in advance and used to control the grid infrastructure later on.

2.3.2 Assets

From the application scenarios outlined in Section 2.3.1, various assets found in Internet computing are deduced, as shown in Figure 2.9 and described below.

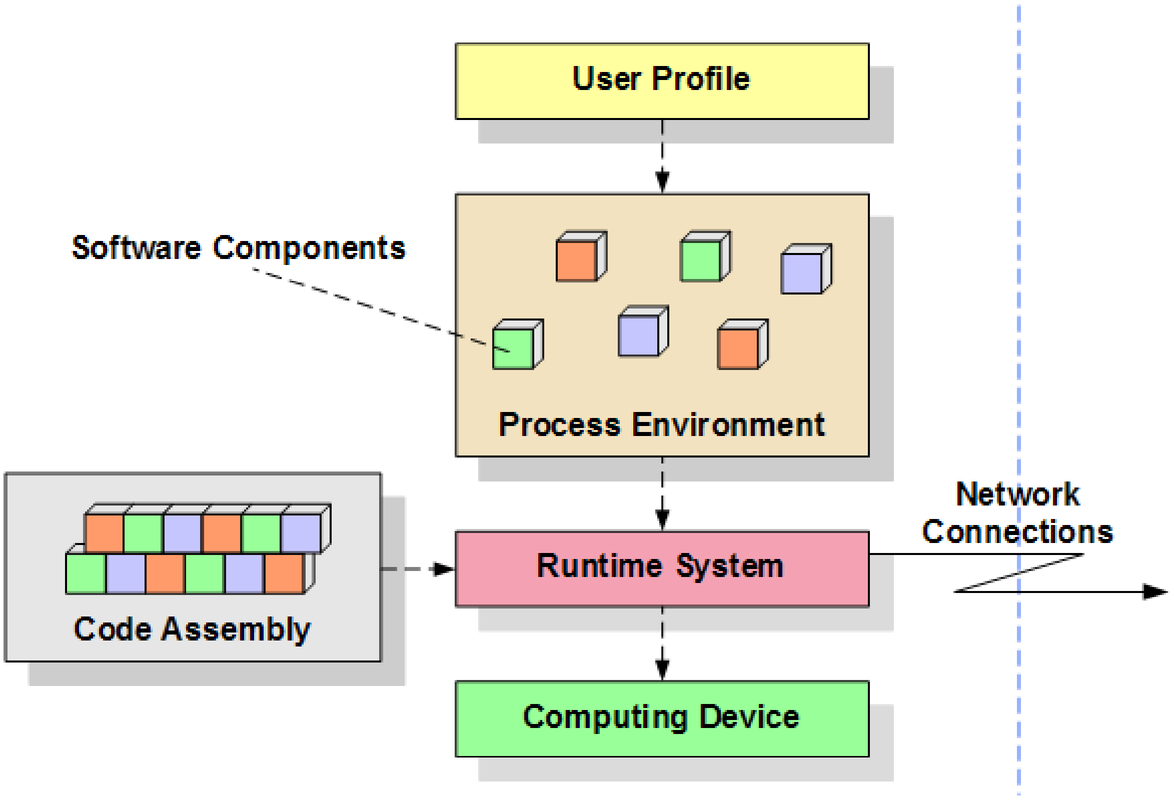


Figure 2.9: Assets of Internet Computing

Computing Device. The first asset is the *computing device* used to manage and run an Internet application. There are various hardware platforms and operating systems, and once installed, they are typically not extended nor modified during runtime. However, an Internet user may switch from one computing device to another or concurrently use different devices, e.g. a notebook and desktop computer while synchronizing his or her meeting calendar.

Runtime System. The next asset is the *runtime system* that actually executes the Internet application, such as Sun Java or a Perl interpreter. It can either run various applications at the same time or exclusively be used for a single application only. Particular runtime environments, such as applet or servlet containers, require extra application frameworks that offer advanced features like dynamic service loading and multi-application resource management.

Software Component. An Internet application is typically built from diverse *software components* that are appropriately composed to form the desired application functionality. Some components may be dynamically loaded and shared by concurrently hosted applications while other components are exclusively used by a single application. Particular component approaches like Enterprise Java Beans (EJB) need an appropriate application container to run, e.g. offered by an application server.

Code Assembly. Software components are usually not deployed individually but grouped and distributed using a *code assembly*. Popular examples of Java code assemblies are Java Archives (JAR), Web Archives (WAR) and Enterprise Archives (EAR). Though code assemblies may be separately created and managed, they usually rely on a common specification that allows the runtime system to exchange and combine them by request.

Process Environment. If an Internet application is started, a related *process environment* is created that contains the application components and the task data. Depending on the underlying runtime system, multiple applications may be also concurrently hosted in a single process environment, each in a separated thread environment. This is often used for server-side Internet applications operating as services for handling a user request.

User Profile. The execution of Internet applications may be customized by a user to meet his or her preferences. The related settings are stored in a *user profile* that is evaluated when a process environment associated with the corresponding user is created, e.g. by selecting the look-and-feel of a GUI desktop. A user can create different user profiles for distinct computing systems or use a single profile that is synchronized among the employed computing devices.

Network Connection. The final asset in this enumeration is *network connections* used by Internet applications to access remote resources. Depending on the application scenario, various middleware approaches are used, such as service-oriented, message-oriented and object-oriented middleware. From a user's perspective, the use of remote resources is often transparently hidden by the underlying middleware approach.

2.3.3 User Roles

Various users are involved to set up, maintain and customize an Internet computing system as well as to develop and deploy an Internet application and its software components, as shown in Figure 2.10.

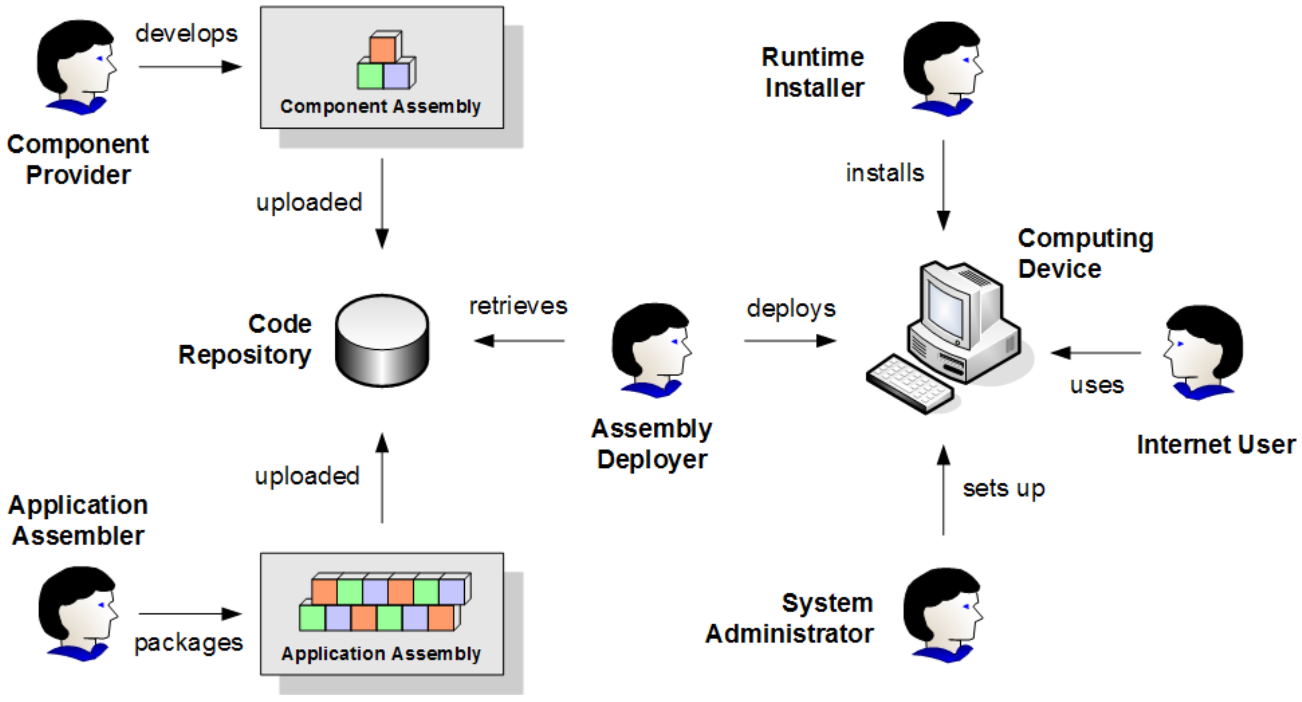


Figure 2.10: User Roles in Internet Computing

System Administrator. The *system administrator* is responsible for the basic setup of a computing device. Typically, he or she initially installs the hardware components and the operating system. The continuing maintenance and update of core features, such as device drivers and hard disks, are further tasks. In our consideration, the system administrator is not involved in installing and configuring a specific Internet application.

Runtime Installer. Another role is *runtime installer* that denotes users capable of preparing a suitable runtime for hosting Internet applications, e.g. installing a Java Runtime Environment (JRE) or downloading and configuring a Perl interpreter. The installer has to consider the current system setup and choose appropriate software packages needed to execute the supposed Internet applications, e.g. setting up an application server to host Java servlets.

Component Provider. The *component provider* is a developer who works on the development of distinct software components. They are encapsulated in a component assembly and uploaded to a code repository. In this context, a component assembly is a particular code assembly that does not contain an entire application but components along with related specification, e.g. about required runtime properties and dependencies to other components.

Application Assembler. Similar to the tasks of a component provider, an *application assembler* works on the development of applications by selecting software components and packaging them in an application assembly. An application assembly does not have to contain software code but may also refer to component assemblies containing the desired components, e.g. by using query statements to look up for components matching a certain release.

Assembly Deployer. The *assembly deployer* is the link between the development and the execution of an Internet application. He or she decides which applications to deploy on a specific computing device and manually retrieves and installs related assemblies from the code repository. In a typical scenario, there are various repositories to choose from and the deployer has to monitor the repositories for recently released software updates and bug fixes.

Internet User. The final role is the *Internet user* who is actually employing the computer device to launch a specific Internet application. The required runtime and application components are already installed and configured appropriately. In addition, the Internet user may customize the existing configuration and store his or her preferences in a user profile. Besides a human user, an Internet user may also be an application accessing the computing device via a network interface.

2.4 On-Demand Internet Computing

In this section, the definition of *On-Demand Internet Computing (ODIC)* is deduced by considering the shift from resource-centric to task-centric computation introduced with regular *On-Demand Computing (ODC)* in a well-known network environment like an Intranet or an Extranet. Afterwards, on-demand facets of ODIC are worked out with particular respect to the elaborated assets and user roles of Internet computing in the previous chapter.

2.4.1 From Resource-Centric to Task-Centric Computation

There are a number of visions concerning ODC that are illustrated by means of different business cases and application scenarios, such as nomadic computing, utility computing and cloud computing. As summarized in [110], *"on-demand computing is a broad category that includes all the other terms, each of which means something slightly different"*. Depending on the concrete use, particular features are emphasized in one scenario while in others they are not considered, such as adaptive bandwidth allocation and dynamic software composition. A common characterization follows the use cases that have emerged along the rise of ODC and are described below.

Infrastructure Management. Inspired by the success of web hosting and virtual private servers, an increasing number of companies discover that outsourcing of infrastructure management reduces costs by sharing commonly used facilities, such as computing centers and backup systems. In this scenario, on demand computing is also interpreted as an approach to facilitate the management and utilization of own resources across distinct parties in the same company. The costs are shared on the amount of resources actually consumed and therefore reflect the real level of business activity. A typical example is the dynamic allocation of multiple cluster nodes according to the current task requirements. In recent time, this trend is also denoted as *Infrastructure-as-a-Service (IaaS)* [218].

Application Server. A refinement is the exploitation of managed computing platforms with an integrated application stack already installed. The deployment of custom applications is facilitated by removing the need to install, configure and manage common software solutions stacks, such as a LAMP installation (Linux, Apache, MySQL, and PHP), a Java Servlet Container or an EJB server. Typically, the application servers are hosted in a computing center and are remotely accessed, e.g. by using SSH. In such a scenario, the idea of providing pre-configured application servers has been further developed towards *Platform-as-a-Service (PaaS)* where the complete application development and deployment are exclusively performed over the Internet, such as with Google AppEngine.

Service Provider. The next step is the utilization of ready-to-use appliances provided by service providers, e.g. a web hosting platform or virtual root servers running on top of a shared computing platform. A further development is the invention of a multi-tenant architecture in which distinct customers share the same computing resources, e.g. database and process environment. While customer-related resources are virtually separated by the shared application instance, the overall amount of memory and processing overhead is reduced compared to launching and terminating isolated application instances for every customer. In particular, this approach has gained much attraction for accessing Internet application services and is denoted as *Software-as-a-Service (SaaS)*.

Software Leasing. In traditional computing, users buy software packages and a number of related licenses for lifetime; regardless whether the software is still needed or not, e.g. after the end of a project. In a particular use case of on demand computing, required software is not bought but leased for a period of time which typically cuts down associated license costs. In addition, a user can select which features of an application suite he or she actually needs and pay only the requested ones. The software is then executed on a remote host, e.g. an application service provider, or the components are downloaded and only runnable during the time period paid for. Following the naming schemes of IaaS, PaaS and SaaS, this trend might be denoted as *Component-as-a-Service (CaaS)*.

Common to the illustrated use cases of ODC is the shift from *resource-centric towards task-centric computation* replacing static resource allocation by dynamic resource assignment, as shown in Figure 2.11.

In traditional resource-centric computing, a computing resource is prepared by an administrator to perform certain types of tasks only, e.g. number crunching or video conversion. In task-centric computing, a user issues tasks and uses various computing resources without task-specific setup. Since there is no static task assignment, a resource can be no longer appropriately administered and provided for every possible computational task by the administrator in advance. In contrast, various resources, such as application configurations, software components and user profiles, may be used to perform the computation; and have to be provided on demand and managed in the moment the task is issued by the user.

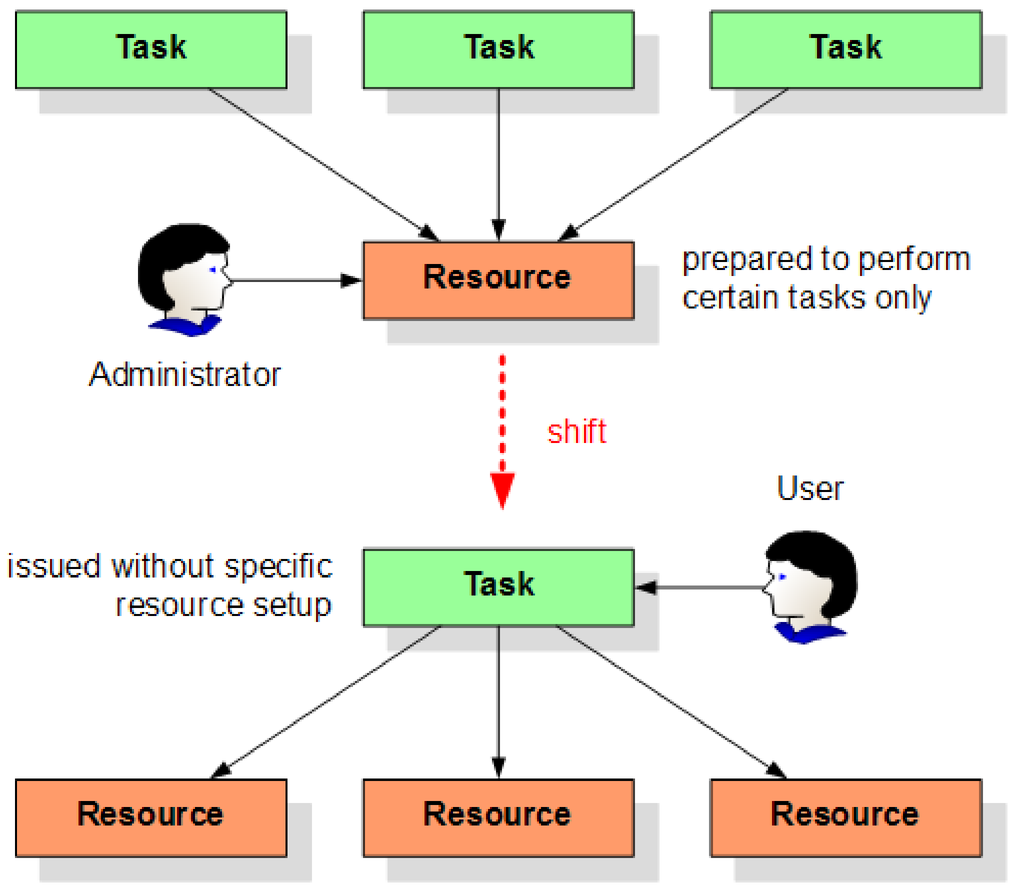

Task
Task
Task
Resource
Administrator
prepared to perform certain tasks only
shift
User
issued without specific resource setup
Task
Resource
Resource
Resource

Figure 2.11: Shift from Resource-Centric to Task-Centric Computing

2.4.2 Facets

Transferring the ODC approach to the Internet environment, *On Demand Internet Computing (ODIC)* is defined as the spontaneous provision and configuration of Internet assets to perform a computational task on demand. Along the assets illustrated in Section 2.3.2, different facets of OD-IC are distinguished as outlined in Figure 2.12 and described below.

Device On Demand. A basic facet is the dynamic allocation and grouping of computing resources. A resource is no longer tightly bound to a certain task but is dynamically allocated and used in different scenarios. For example, a company may offer computing time in a high-performance server farm according to the Grid approach. A customer can choose the number of servers that should be utilized to process a complex computation in a given time. A metering service turns the servers into utilities on demand that are only paid for the time they are used. In another scenario, desktop computers may not always be utilized and could be virtually combined to assist in a global computing effort like in the *SETI* and *Folding@home* projects [11, 115].

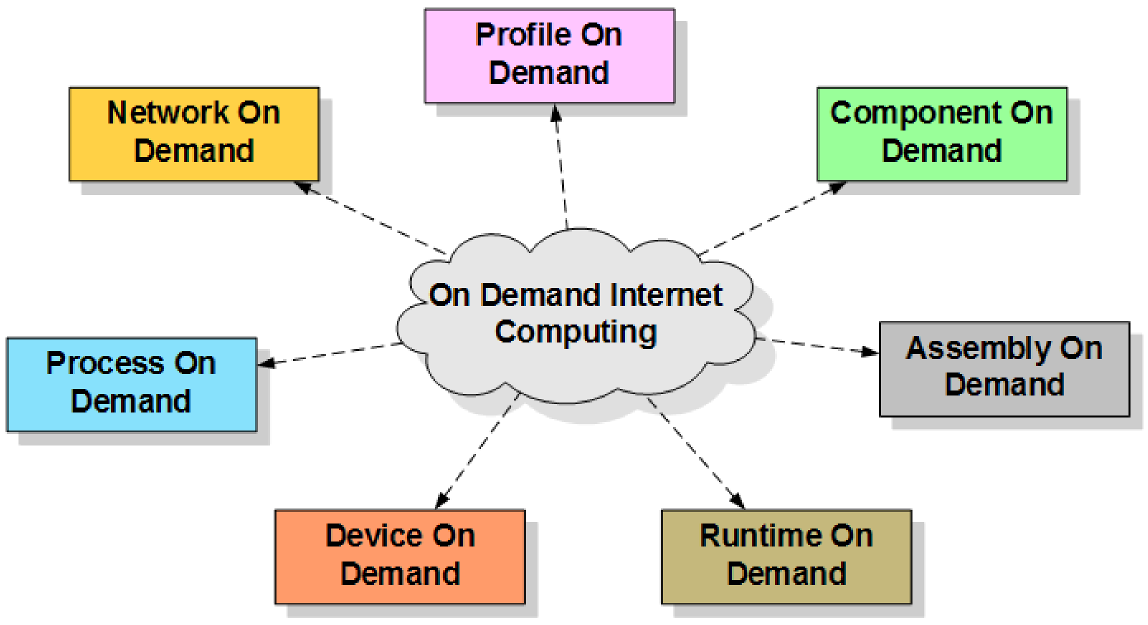


Figure 2.12: Facets of On Demand Internet Computing

Runtime On Demand. There are various types of Internet applications that need different runtime environments, such as Sun Java Runtime Environment (JRE), Microsoft Common Language Runtime (CLR) or a Perl interpreter. The shift towards ODIC requires to provide appropriate runtime environments on demand. On the client side, popular examples are browser applets, such as Flash animations, which offer to automatically download and install the required runtime before starting the actual applet. The user does not have to ask an administrator for manual installation, and the runtime loaders are often designed to choose the proper installation files for the currently employed computing device and operating system.

Assembly On Demand. Besides the spontaneous employment of an available computing device and the automatic installation of the runtime environment, the application itself has to be deployed and configured. This is often achieved by using remote code repositories and deployment units that can be easily retrieved from the requesting computing node. For example, a Java applet is linked in a web page along with the URL referring to a related Java Archive (JAR) on the server. The JRE plugin of the Internet browsers downloads the JAR file as soon as the web page is visited by the user and starts the Java applet afterwards. In this example, there is no need to install software packages in advance. This allows users to virtually start every applet on every computing device.

Component On Demand. Another on-demand facet is the provision of application components and the dynamic creation of custom applications by assembling separately developed and deployed components. For Internet computing environments, this is typically achieved by using particular dynamic code loading approaches, such as MS ActiveX, Sun JNLP and OSGI. They allow retrieving application components from code repositories via the Internet and composing the application during runtime. The sharing and reusing of components between different applications does not only allow reshaping computing systems according to the specific task and resource capabilities but also saves resources, such as network bandwidth and computer memory.

Profile On Demand. A further facet is the seamless customization of software and resource assets according to the needs of the employing user. This is essential for software installations and resource elements that are not used by a single user only but shared and alternately applied like in the Internet. As an example, a desktop computer in a public space, such as an Internet cafe or a library, may be used by different users, each launching a locally installed Internet browser. Although not using their own computer, every user expects to work with his or her personal bookmark list, while other users are not allowed to gain access. The profiles are applied without human intervention and used to seamlessly customize the currently employed computing device.

Network On Demand. In contrast to a well-known and managed network environment, such as an Intranet, the location of remote resources in the Internet may be unknown. Similarly, a connection to a resource can be only set up if client and server use the same protocol configuration. To this end, a resource registry, such as UDDI for web services or a naming service for CORBA, is introduced which allows client and servers to register and query resources, respectively. In addition, some approaches offer to download and include a suitable stub for connecting a remote resource on the fly, e.g. *Sun Jini* [373]. The overall idea is to bridge the network without bothering the user or the application to do so.

Process On Demand. The final facet in this enumeration is the movement of a process from one computing device to another, such as found in mobile code scenarios. For this purpose, the process is typically suspended; its data is packaged in a transferable form and transmitted to the remote runtime environment that has been appropriately prepared. The application code is either transmitted as well or retrieved by the remote node. After all, the resource bindings are re-established and the process resumes its computation. This facet allows users to switch computing devices while applications are still in use and enables to balance the overall load of a computing network without task interruption, e.g. due to server shut down or maintenance work.

2.4.3 Vision

The shift from resource-centric to task-centric computation will let computing devices respond faster, operate in a more adaptive manner and open new ways to perform specific user tasks. Following this vision, ODIC will handle uncertain task requirements and heterogeneous platform configurations to run Internet applications on distinct Internet computing devices, as illustrated in Figure 2.13.

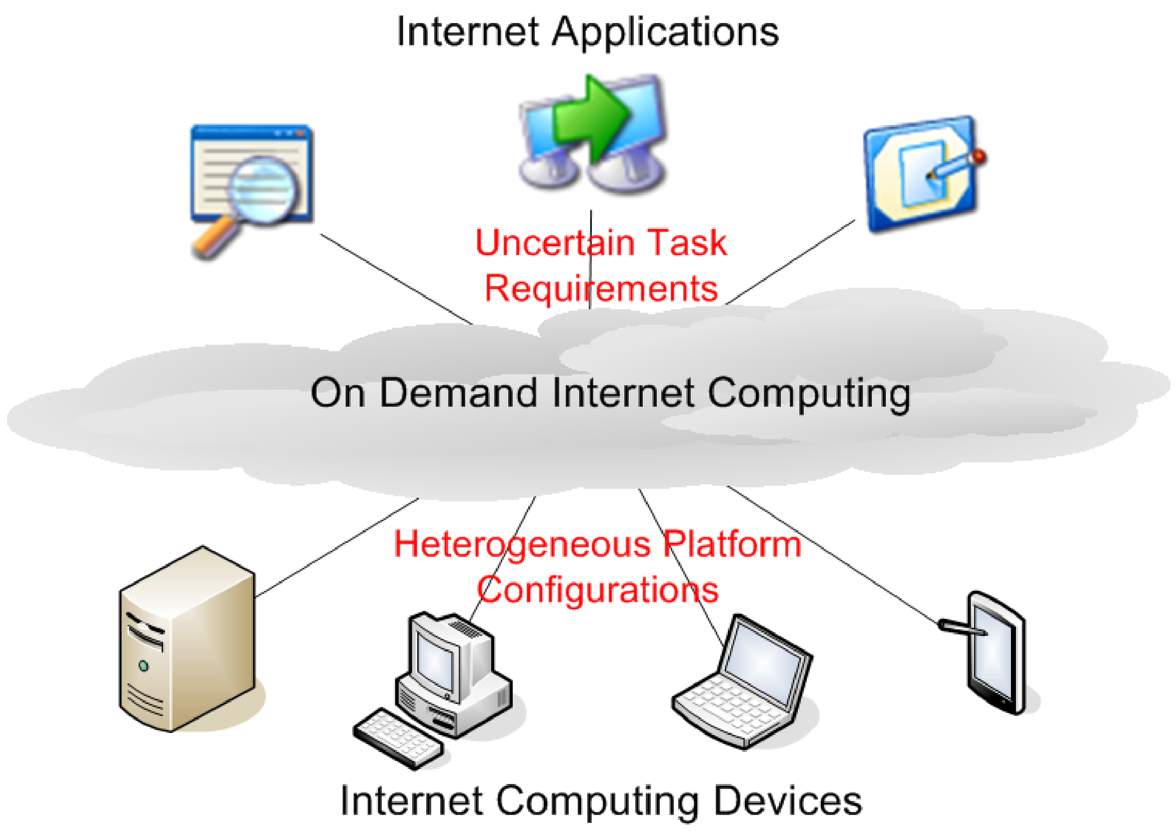

Internet Applications
Uncertain Task Requirements
On Demand Internet Computing
Heterogeneous Platform Configurations
Internet Computing Devices

Figure 2.13: Vision of On Demand Internet Computing

The fixed coupling of application installation and computing device is replaced by a flexible processing setup performed on-demand. Concerning the computing scenarios presented in Section 2.3.1, various applications of ODIC are envisioned, as outlined below.

Local Task Processing. The first vision is the support of local task processing by using a GUI-based *pervasive application workbench*. It hides the concrete implementation and composition of the Internet application as well as the configuration and customization of the local computing device. This will allow users to move from one computing device to another while being able to access their information and applications in a uniform and pervasive way.

Remote Task Processing. The ability to deploy and launch Internet applications on demand will be used to turn computer devices into *shared computing factories* and to support remote task processing in a multi-tenant manner. Related application environments are appropriately installed and shared by various processes without administrator intervention. Common resources, such as software components and network connections will be dynamically assigned and utilized.

Distributed Task Processing. Another vision is the association of heterogeneous computing devices to a *public platform federation* and the disappearance of the individual Internet computing device used to process a task. Task processing requests are deployed to the federation without knowledge about the actual setup and may be distributed to one or many computing nodes. Related task applications and data are transparently moved within the platform federation.

2.5 Related Approaches

In this section, well-known approaches of Internet computing are discussed and how well they address the presented facets of ODIC to support the shift from resource-centric to task-centric computing. Based on the distinction of the presented vision in Section 2.4.3, the regarded approaches are categorized into *local task processing*, *remote task processing* and *distributed task processing*.

2.5.1 Local Task Processing

In local task processing, a computational task is performed on resident computing systems and the related software components are dynamically retrieved and run the time the task is issued. In contrast to remote task computing [see Section 2.5.2], no task data has to be transferred to a foreign and potentially unsafe node, e.g. local task computing may access a local database where remote task computing is not suitable.

Thin Client Computing. Since the web computing approach utilizes the omnipresent web browser to enable the ubiquitous access to remote Internet resources, it also forces Internet applications to use a limited HTML/HTTP based user interface. This has lead to the development of browser plugins that can be added to the browser installation. They offer to run a custom Internet application, also known as applet, within the browser process environment, as shown in Figure 2.14.

The basic idea is to allow the deployment and running of custom Internet applications on the client side in the same way as web pages are downloaded and rendered by the browser, e.g. by integrating the application interface into the web page as with Java applets and Flash animations. To this end, specific browser plugin are installed in advance. They do not contain the Internet application themselves, but create a suitable application container whenever a related applet is downloaded and about to run. With custom client-side applications, the service does no longer have to deliver standard HTML pages but may rely on proprietary application communication, e.g. transmitting a 3D graphic model that can be modified by the user via an advanced applet control element.

Although the thin client approach offers flexible visualization and interaction features almost like a regular desktop application, it still relies on the server-side business logic code. Similar to web pages, applets can only be used while they are connected to the remote server. They are not designed to run outside the browser. Typically, there are restrictions for the applet to access sensitive data such as local files, or to connect to a remote server other than it has been loaded from such as in the sandbox model of Java. A further drawback of the thin client approach is that it introduces a particular programming and runtime model that makes it complicated to transfer a regular application into an applet. In addition, concurrently running applets are usually hosted in distinct runtime environments and do not share client resources or program code.

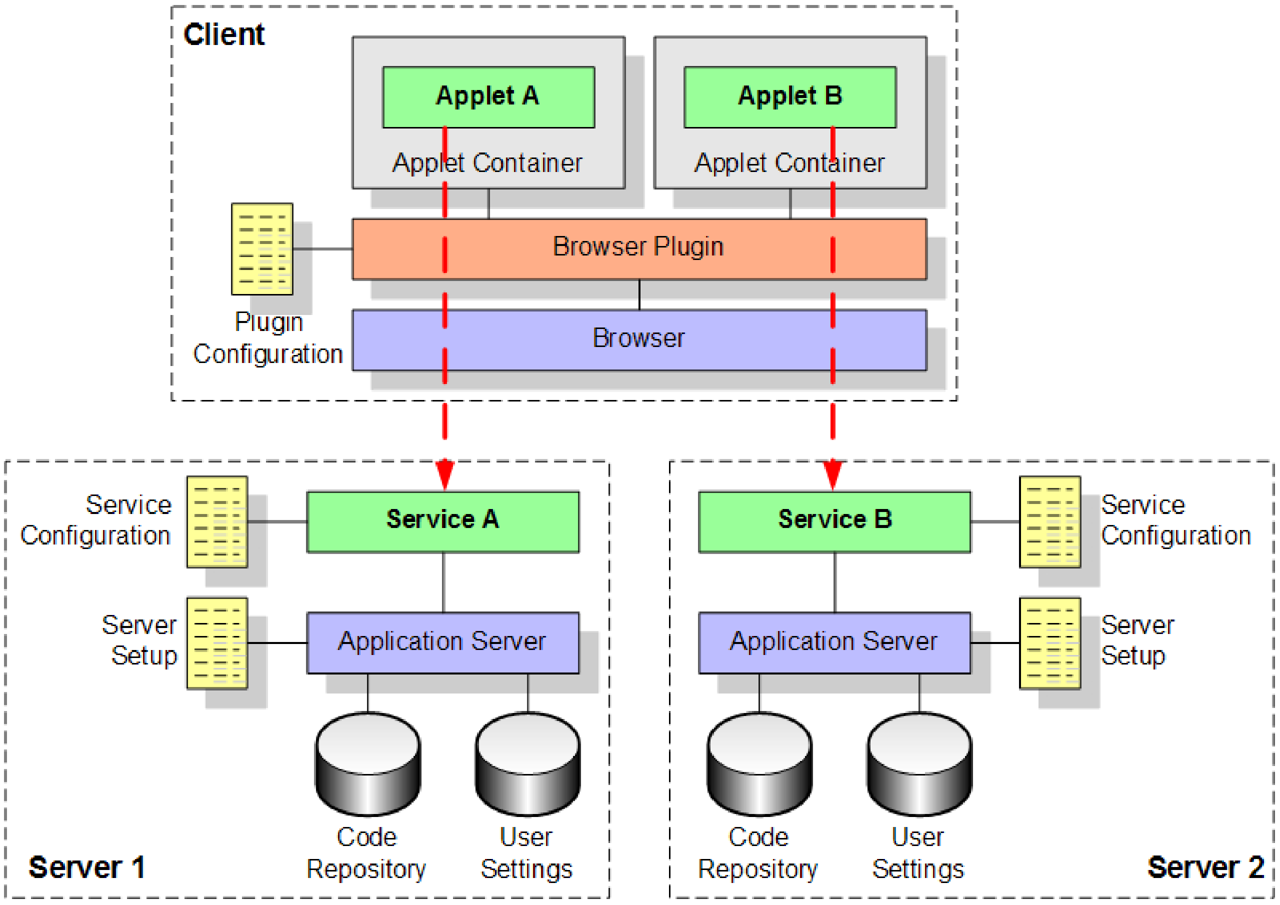


Figure 2.14: Thin Client Computing

Concerning ODIC, the thin client approach already enables the spontaneous execution of custom Internet applications on the client side. It overcomes the limitation of the native web computing approach without bothering users to manually install a client application for every service, as in regular service computing. The required browser plugins are often available for a broad range of computing systems and typically introduce a virtual runtime environment that frees application developers to create different applets for every target computing system. In summary, thin client computing is a feasible approach to deploy custom application interfaces for accessing specific remote services. However, there is no support for deploying entire applications on another computing device, e.g. for local task processing in a personal computing scenario.

Rich Client Computing. A further step in Internet computing was the introduction of the rich client computing approach that addresses the deployment of Internet applications on the personal computing device of the customer. For this purpose, a Rich Client Framework (RCF) is installed on the potential clients. It controls an application environment on top of the native operating system to run custom Internet applications, as shown in Figure 2.15.

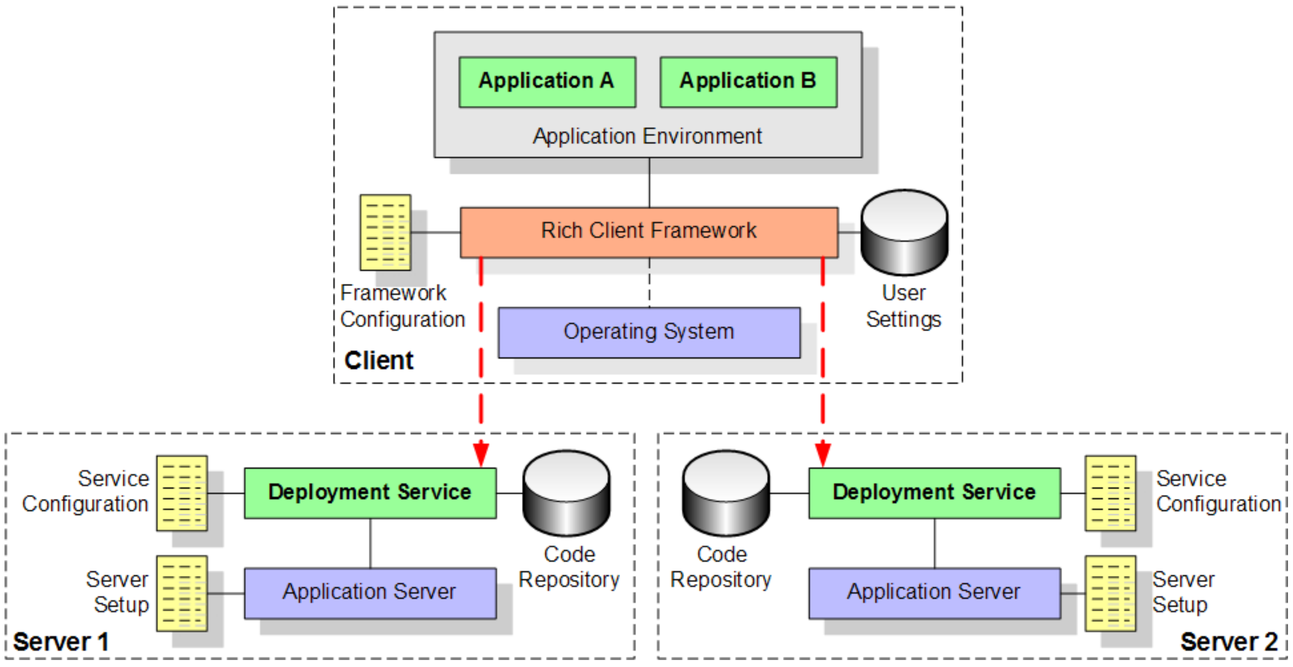


Figure 2.15: Rich Client Computing

Since the RCF should be able to manage multiple applications, rich client applications are designed to share client resources and program code. They are deployed using a component deployment model. The RCF caches downloaded deployment units and reuses compatible components if an application is to be launched. Missing and updated components are requested from remote deployment services and retrieved automatically. In this context, the RCF itself often provides particular features to the applications such as an integrated user interface that can be shared by the hosted applications, e.g. a common application menu or window control. Applications may also use components from other applications to extend their original functionality, e.g. loading a new version of an image viewer plugin. In contrast to thin client applications, a rich client application does not rely on remote services and can be used offline as well. The rich client computing approach has gained much attraction with the advent of advanced RCF implementations, such as Eclipse Equinox [95]. The possibility to deploy and update fully-fledged applications across the Internet, such as with Java Web Start, has simplified the deployer's work and represents a step towards separating the fixed tangling of computing device and application installation. Moreover, the manual and shared installation of the RCF enables to adjust the framework exactly to the capabilities of the underlying computing device and operating system. This moves this task away from the applications. On the other hand, rich client applications often have to follow a specific programming and deployment model that incurs additional effort to the application developers and deployers. The setup and maintenance of distributed deployment services is still subject of ongoing development, e.g. with respect to reliability and synchronized code distribution.

The rich client computing approach addresses several features of ODIC, such as the spontaneous deployment of software components to the client. By using a RCF and remote code repositories, an application deployer is no longer needed. If the repositories are linked and assemblies are exchanged on request, application assemblers and component providers do not have to upload the assemblies to each repository. In contrast to the previous approaches, the RCF may run Internet applications not only online but also offline. A major drawback of this approach is the focus on a single device installation that requires installing and setting up custom applications on every participating computing device. This is also valid for the user settings that are typically not synchronized across the employed computing devices.

2.5.2 Remote Task Processing

In remote task processing, the utilized computing resources do not reside on the local computing device but on distinct remote computing systems that are typically shared with other users. A computational task is deployed to usually one server and the results are sent back to the client after completion. To this end, an appropriate task processing service is installed and permanently run on the remote site while a control application on the local site can connect to the service and does not have to wait for the task results, e.g. when retrieving them in an asynchronous manner.

Service Computing. The first regarded approach is *service computing* in which resources of remote computing devices are utilized to perform a specific task in place of the issuing computing node. In this scenario, there is a client machine that is used to prepare a task and to interact with the remote server machine connected via a network link, as shown in Figure 2.16.

The actual processing is done on the server machine on which a corresponding service has been installed and configured before. To this end, the server administrator has to set up an application server (Server 1) and to configure a set of services according to the purpose of the server, e.g. performing video processing or data mining. In a typical scenario, a local code repository is used to store related code components in well-known locations from where they are retrieved when the service is started. The same procedure is done for every other server (Server 2). The client is typically a regular computing device, such as a desktop computer, which is also set up in advance and installed with appropriate applications suitable to access the desired remote services, e.g. via web services or proprietary network protocols. The user settings for customizing the client computing environment and service operation, such as account credentials, are partially managed on the client and on the server side.

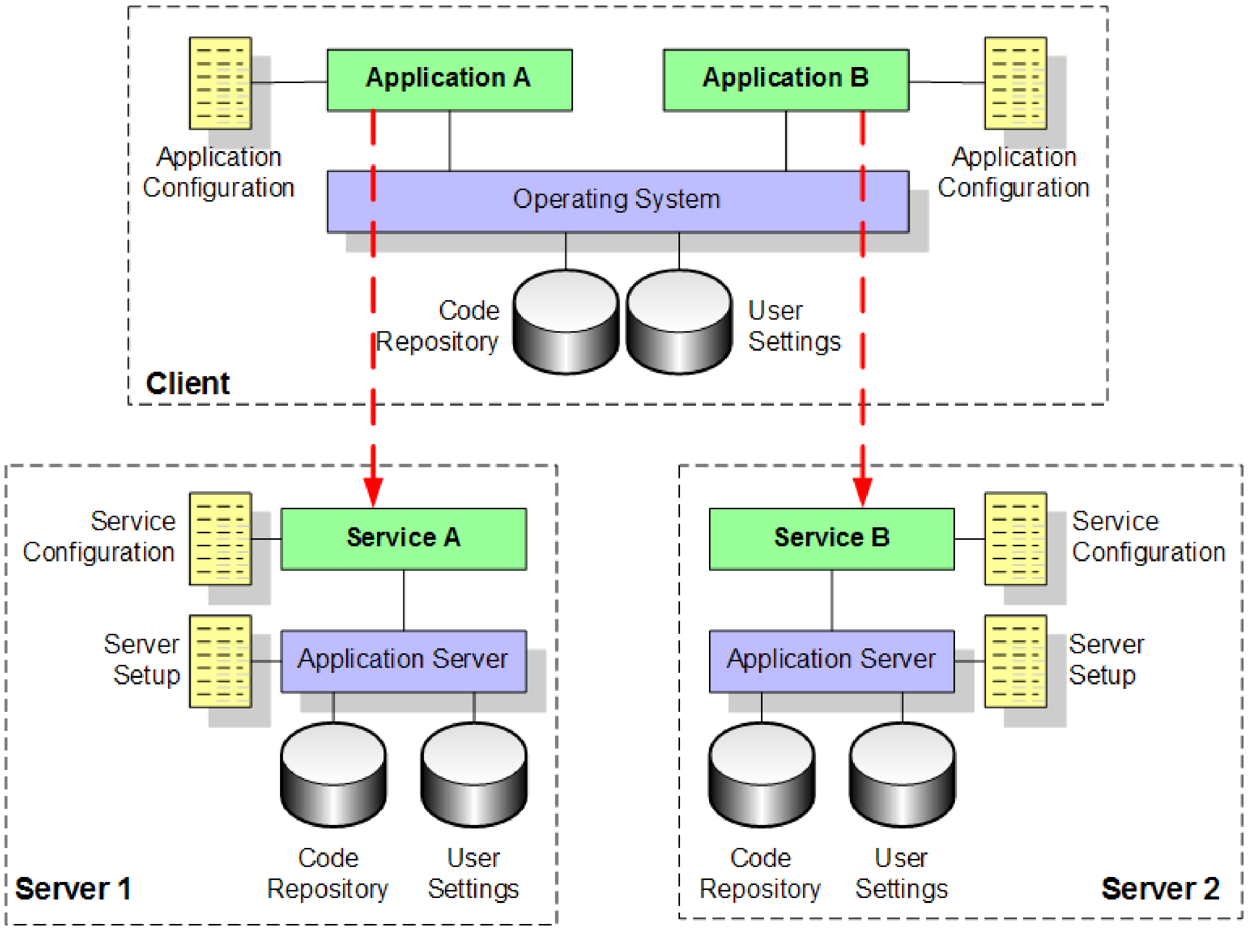


Figure 2.16: Service Computing

The major advantage of this approach is the decoupling of task preparation from task processing. This is achieved by using services that are uniformly accessible although they are managed separately. The actual implementation of the client and service instances is hidden, which supports operating in a heterogeneous environment, e.g. C++ clients can transparently access Java services. A service registry can be used by service providers to announce the availability of specific services. In turn, clients may query the registry for suitable services according to their attributes, e.g. costs and quality of the services. Another advantage is the simultaneous processing of parallelizable subtasks by concurrently issuing requests to various service instances.

Concerning ODIC, the online-only operation mode of the service computing approach limits its use to remote task processing. It does not support mobile applications and seamless customization among different computing devices. A service can only be used if a suitable client application has been installed, and the spontaneous deployment of services to further computing devices is not supported. In practice, this is complicated by using different code repositories on every site to which the service components have to be deployed. Furthermore, the user settings to personalize the client application and the services are separately managed and must be synchronized by the user, e.g. switching from one service to another, the settings are not transferred but the user has to customize the new service again. In addition, service computing still involves various user roles, such as platform administrator, runtime installer and application deployer. Thus, service computing is not suitable for ODIC with respect to running a random application without manual user intervention.

Web Computing. An approach similar to service computing is web computing that is also based on the remote execution of a service but uses a regular web browser on the client side. In this scenario, the web browser is used to offer a uniform and standardized graphical user interface to different remote services without the need to install a custom client application for each type of service, as shown in Figure 2.17.

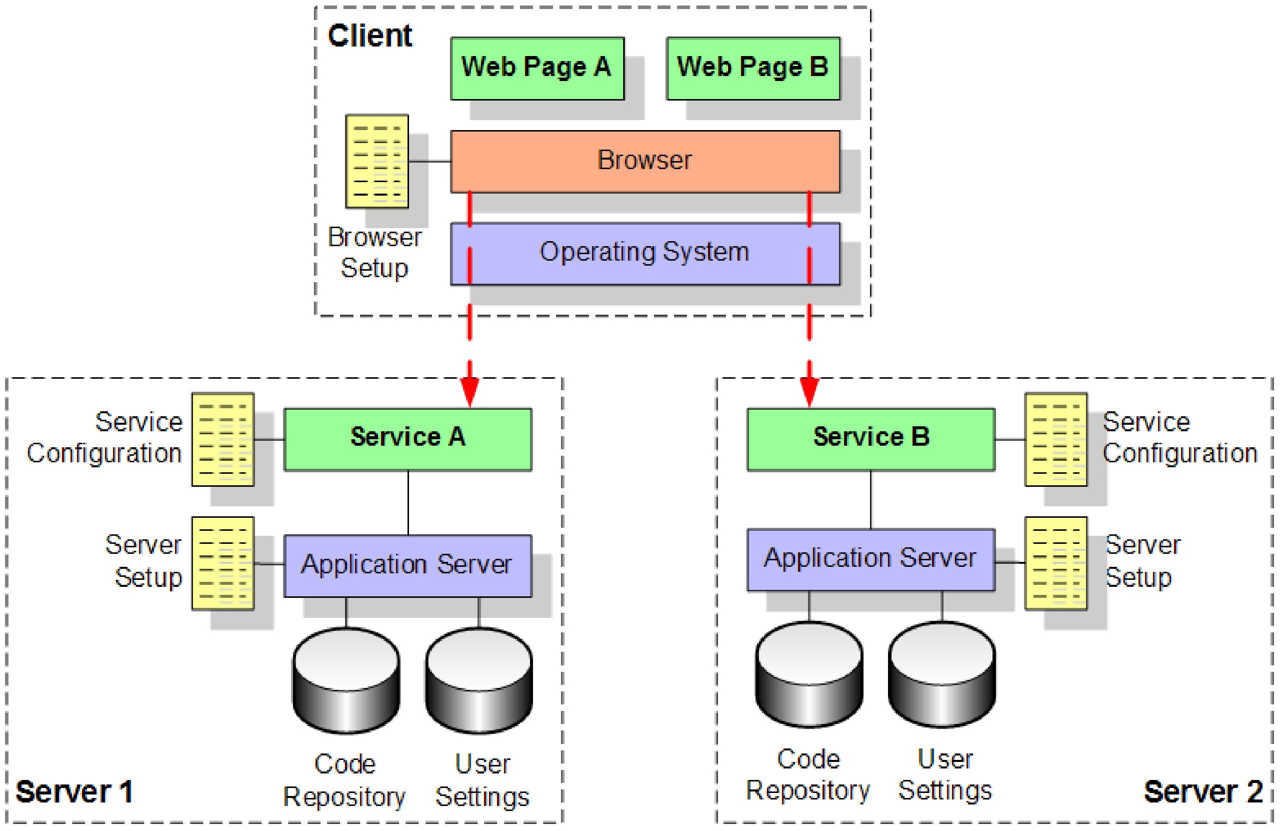


Figure 2.17: Web Computing

In a typical web service environment, the application server offers a standard runtime environment for hosting web servlets that may be accessed via regular HTTP, e.g. Jakarta Tomcat [16]. A new service is simply installed by deploying the servlet assembly to the application server, e.g. a web archive (WAR) or an Enterprise Archive (EAR). Afterwards, a suitable servlet environment is automatically created in which the service is hosted and run; typically without interfering with other servlets. In addition, the server administrator may install and configure common resources, such as a database connection pool, which may be shared by the servlets. On the client side, the web browser does not need further maintenance once it has been installed. In contrast to native service computing, there is no code deployment to the client side since the browser does not execute specific business logic but acts as a user interface that renders common HTML elements.

The web computing approach owes its success to the low-profile requirements of a regular web browser that is installed on nearly every personal computing device today. Consequently, related web services may be accessed from anywhere and at anytime. This commoditization enables new ways of interacting with remote resources, e.g. blending the access on distinct services into a common web interface like in Internet portals [378]. Another trend is the activation of the formerly static web page by using browser inherent features, e.g. by using Asynchronous JavaScript and XML as in AJAX [293]. The web page is no longer entirely loaded but only the parts currently needed by the user. The remaining components are downloaded in the background and separately updated which results in better overall response of the web interface as in the *Rich Internet Application* approach [224]. The emergence of web application frameworks like Java Server Faces [345] has simplified the development of web portals that impersonate the traditional graphical desktop computer interface. Particular code migration tools like the *Google Web Toolkit (GWT)* allow developers to write a desktop application by using the Java programming language and to compile it into an AJAX application [298, 144]. The Web 2.0 approach aims to change the web at all by including users to create, customize and offering web content and services, e.g. combined in so called *service mashups* [31]. To this end, service interfaces are opened for public access, such as the Google search API, and related services can be used in other services and included in private web pages.

The basic characteristic of web computing featuring ODIC is the remote execution of the actual application and the resulting dependency on an Internet connection. In this context, the transmission of sensitive data or personal documents may not be possible due to security threads. Although there is an ongoing progress in transforming the web browser to a desktop-like user interface, it is not possible to reuse the same web interface implementation for different services. Furthermore, web computing does not address the spontaneous deployment of remote applications and does not maintain user profiles among different service instances.

2.5.3 Distributed Task Processing

To perform a specific task, remote and local task processing approaches are limited in the number and types of computing resources to select from. An advancement is distributed task processing where a task is split and concurrently executed on various computing nodes. From this point of view, task data and required software components are distributed across the network.

Peer-to-Peer Computing. The presented local and remote computing approaches separate computing nodes into clients and servers that allocate and offer resources, respectively. In peer-to-peer computing, this is replaced by computing nodes that are explicitly supposed to act as clients and servers at the same time, as shown in Figure 2.18.

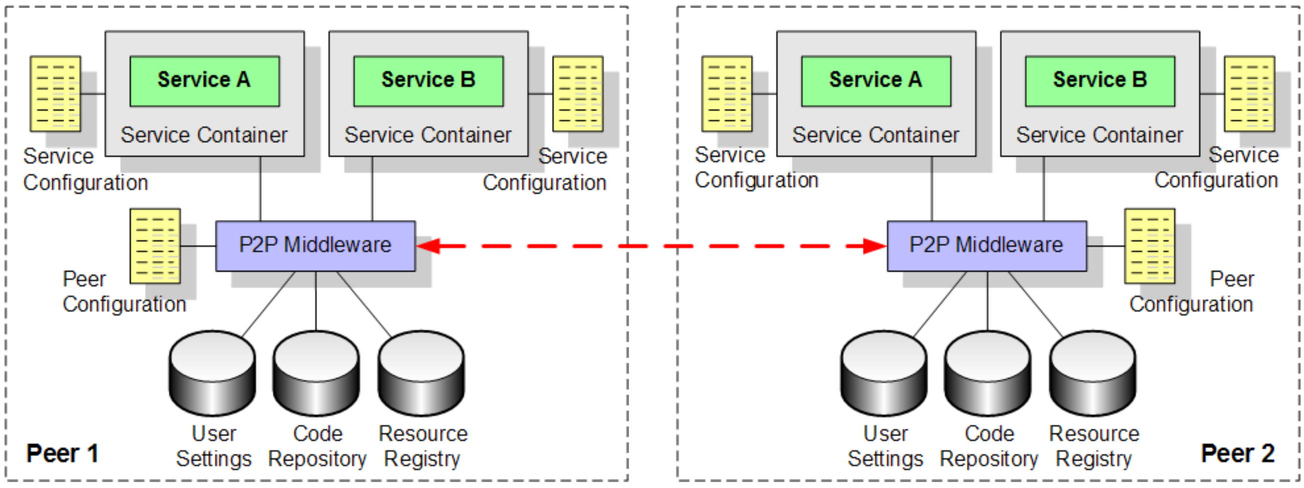


Figure 2.18: Peer-to-Peer Computing

Although a peer may run alone without connecting to any other peer, it usually joins a specific peer network to contribute to a dedicated computation, e.g. offering data sharing services like in the *bittorrent* network. However, there is no remote management of peers in terms of controlling service provision and utilization. On the contrary, a peer may come online and go offline any time, offer services and abandon operation unexpectedly. In a typical scenario, one or multiple peer services are installed on top of a P2P middleware that handles common peer service functions, such as peer discovery, service announcements and network communication. In this context, a peer network is usually limited to computing nodes using the same P2P middleware. In fact, a peer may only join a P2P network if an appropriate P2P middleware has been manually installed before. The P2P computing approach has gained much attention with the rise of file sharing systems [60], such as Napster and Gnutella [303]. Other popular applications are the support of complex computations, such as in SETI [11], and the provision of novel collaboration tools like instant messengers [121]. A major benefit of P2P systems is the self-managing organization of resource announcement and discovery [287]. This has also enabled the peer integration of temporarily connected computing devices, such as personal computers that are likely to disappear without notice. From this point of view, the operation of a P2P network cannot be determined exactly, e.g. required resources may become unavailable though they are still in use or the utilization of available peer resources is poor. For large peer networks, balancing approaches have to be employed that aim at optimizing the peer network in a self-managing way, e.g. by determining the computational load of the peers and reorganizing the task distribution [236]. Further, the distribution of new services on distinct nodes in a large P2P network is usually not possible without the explicit permission and help of the administrator of the respective peer node. In small network environments, e.g. in an enterprise scenario [61], there are particular P2P middleware solutions like Sun Jini [347] that supports the spontaneous interoperation of computing devices, e.g. running in a clustered environment [219]. It benefits from the cross-platform availability of the JRE [189].

Regarding ODIC, the P2P computing approach allows to connect to dynamically discovered peer nodes in the Internet and allocate computing resources as needed. To this end, the P2P middleware frees developers and users from handling basic peer communication while building a transparent resource network, e.g. by connecting to a well-known super peer node and retrieving all currently running peers on-demand. On the other hand, a P2P network is usually dedicated to certain services, such as file sharing, and cannot be used for different computing tasks [112]. The deployment and composition of task-specific software components on the fly is only accepted from trustable peer nodes, e.g. in a small network environment by using Sun Jini. In this context, typical P2P enabling approaches for the Internet, such as JXTA [140], focus more on network compatibility than code compatibility and define how peer services should interact. This finally leads to heterogeneous computing environments and makes it more difficult to deploy and retrieve software components suitable for every peer on-demand [121].

Grid Computing. A refinement of the P2P computing approach is the use of an application middleware that supports the dynamic deployment, configuration and allocation of services in a networked computing environment in the Grid computing approach, remote computing devices are interconnected and form a collaboration network, as shown in Figure 2.19.

There are different types of Grids, such as Data Grids, Service Grids and Resource Grids. The common focus is on supporting the dynamic allocation and interconnection of Grid nodes according to the task requirements, e.g. enabling high-performance computation using many low-profile computing devices in parallel [122]. A specific Grid middleware, e.g. *Globus* [135], *Unicore* [361] or *gLite* [134], controls the task deployment within the Grid and handles topics, such as allocation of additional nodes, accounting of the utilized resources and granting access to device resources.

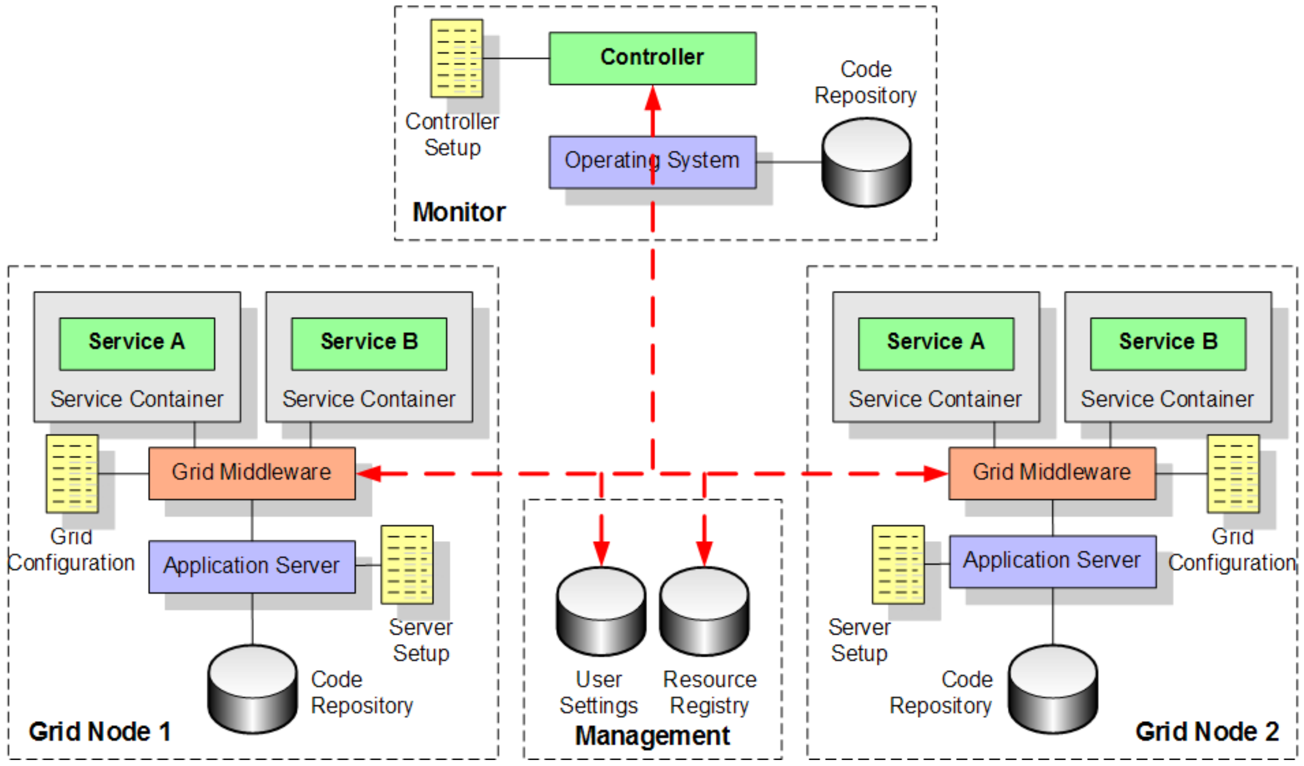


Figure 2.19: Grid Computing

The overall vision of Grid computing is the transformation of networked computing resources into a cross-platform computing environment in which resources are allocated and utilized dynamically. Along with the standardizations efforts to enable intergrid collaboration among competing Grid middleware implementations, e.g. by the introduction of OGSA [261], Grid computing promises to combine remote computing resources as needed without bothering the user to administer the Grid network. However, Grid computing is not designed to handle personal computing applications, such as running a web browser on a local computing device. In addition, running an Internet application in a Grid requires to move the process data to the currently employed Grid node which is not always possible, e.g. in case of a database or sensitive personal documents.

The Grid computing approach represents the state-of-the-art solution to distribute computational tasks and utilize remote computing resources on-demand. Although new services were originally supposed to be installed by the administrator [213], there are approaches that support the hot deployment of Grid services while the application servers are already up and running [125, 326]. In a similar way, a typical Grid installation is supposed to be distributed on well-known and fixed computing systems, e.g. located in an enterprise network, there are approaches to open the Grid idea to on-demand computing scenarios, e.g. by using mobile agents [133] and mobile computing devices [289]. As a result, Grid computing greatly supports the on-demand vision of moving from resource-centric to task-centric computing. However, this is only valid for distributed task processing and does not apply to standalone scenarios, such as found in personal and nomadic computing applications. For example, Grid computing does not address the synchronization of profiles on demand and the spontaneous deployment on yet unknown personal computing devices, such as found in typical Internet computing scenarios.

2.6 Summary

In this chapter, *On Demand Internet Computing (ODIC)* has been introduced from various perspectives. The scope of Internet computing has been outlined by separating it from Intranet and Extranet scenarios and by describing the characteristics of the Internet environment. Then, Internet applications and basic application scenarios have been regarded as well as the assets and user roles of Internet computing. The idea of *On Demand Computing (ODC)* has been presented around the shift from resource-centric to task-centric computing. Afterwards, this idea has been transferred to the Internet, and the facets and vision of On Demand Internet Computing (ODIC) have been elaborated. Finally, the major state-of-the-art approaches in regular Internet computing has been regarded and how well they support the visions of ODIC; namely local, remote and distributed task processing on-demand.

The basic outcome of this introduction to ODIC is the increasing demand to enable regular users to utilize Internet resources for tasks that were uncertain the time the resources were set up. A related proposal is to shift the view from resource-centric to task-centric computation as presented by ODC. The fixed assignment and configuration of resources is to be replaced by *dynamic allocation and customization of resources needed to perform the task*. Although this affects various user roles and creates some effort, it may still be conceivable for a limited number of well-known computing resources, e.g. an administrator group could permanently reconfigure an enterprise computing infrastructure according to changing requests. The Internet, however, represents an open network environment with an unlimited number of separately managed heterogeneous resources that cannot be manually reconfigured to perform uncertain tasks. As a consequence, the static *one-for-all resource administration* has to be changed into a dynamic *configurable-for-all resource automation*. With respect to this assumption, it has been shown that regular Internet computing approaches focus on *individual application scenarios*, address *selected types of computing assets* and reveal a *limited automation of on-demand facets*, as presented in Section 2.3.1, Section 2.3.2 and Section 2.4.2, respectively. Consequently, a common solution is needed that is not limited to either local, remote or distributed task processing, at best considers all required computing assets and enables the common automation of on-demand facets for tasks not known at the time the Internet computing systems were set up.

3. An Autonomic Cross-Platform Operating Environment

3.1 Introduction

In this chapter, the proposal of using an *Autonomic Cross-Platform Operating Environment (ACOE)* to support the visions of ODIC and the required automation of related on-demand facets are presented. The project mission is sketched, and the cross-platform and autonomic operation of the proposed environment is described. The chapter ends with a reflection on existing solutions and a summary of the outcome.

3.1.1 Subject

With respect to the elaboration of ODIC in Section 2.4, the subject of the proposal is the dynamic configuration of networked computing resources to run task applications in various application scenarios by user request, as introduced in Section 1.2.1 and illustrated in Figure 3.1.

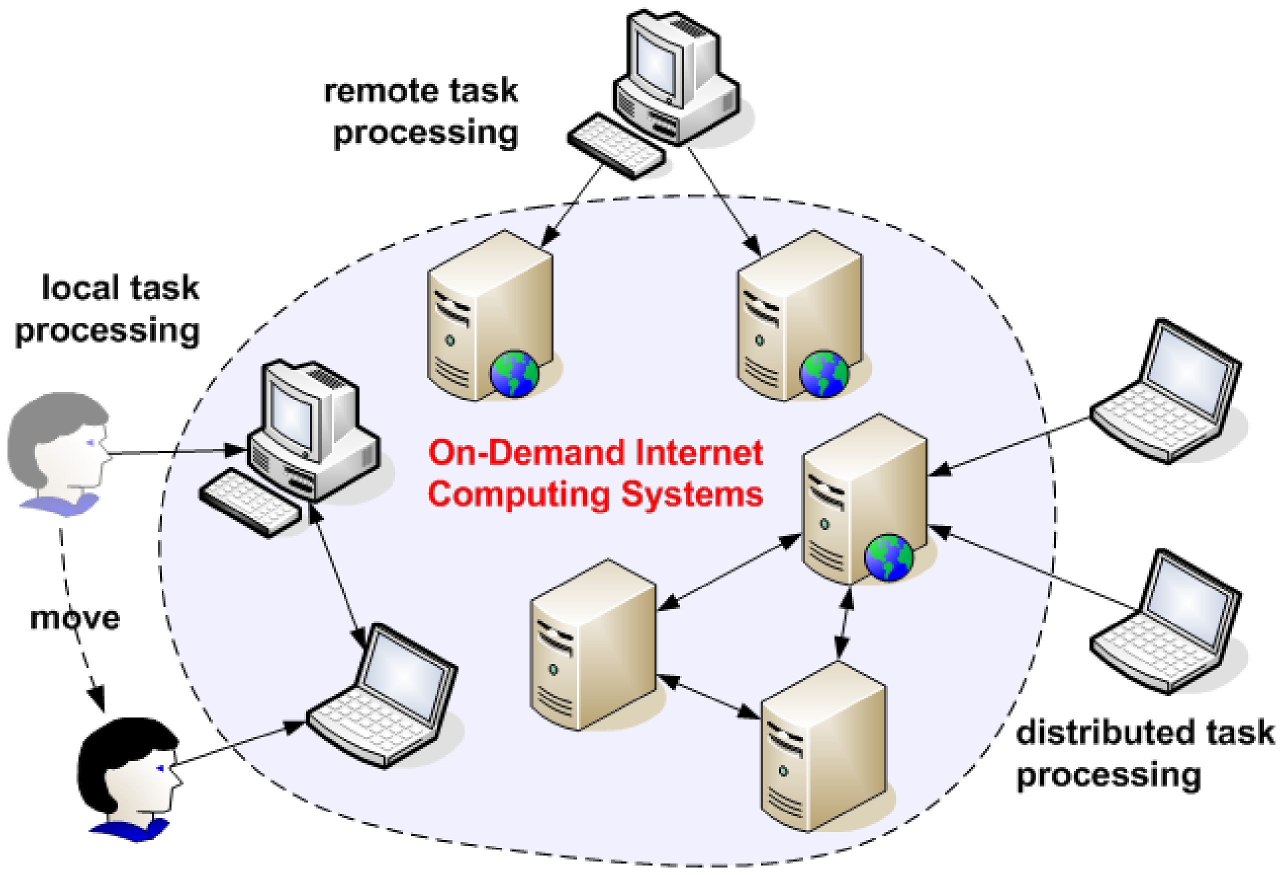

remote task processing
local task processing
move
On-Demand Internet Computing Systems
distributed task processing

Figure 3.1: Running On Demand Internet Applications on Networked Resources

For the use in the Internet, three major issues and affected user roles can be identified that have to be considered by the proposal to support the on-demand vision, as detailed below.

Heterogeneous Computing Platforms. The setup of the proposal involves the use of common Internet computing devices that differ in terms of hardware resources, operating system and platform configuration. In this scenario, *application developers* should not have to consider the setup diversity of personal computers, laptops and server systems as well as of operating systems installations, e.g. based on MS Windows, GNU Linux and Apple Mac OS X.

Uncertain Task Applications. From a general point of view, the computing devices are supposed to host and execute uncertain Internet applications that are not known at the time the devices are set up. The proposed system should free the *platform administrators* to prepare a suitable runtime environment and to retrieve required software components for every application. In effect, it has to be designed to run dynamically requested applications in a self-managing way.

Changing Runtime Scenarios. The alternating and shared use of computing devices is a further issue, in particular if devices and applications are dynamically selected by regular and nomadic users mostly suited for their current task. The proposal should allow *application users* to switch computing devices without the need to manually synchronize their personal settings, application configuration and document data across distinct computing stations.

3.1.2 Idea

The basic idea of the proposal is to decouple the setup of the application environment needed to run the application from the administration of the computing system currently employed by the user. The proposed solution is the establishment of an *Autonomic Cross-Platform Operating Environment (ACOE)* that virtually hides the use of distinct computing systems. To this end, a self-managing integration middleware is installed on every computing node, as shown in Figure 3.2.

Instead of installing and configuring an application on each computing device, applications are deployed once and are enabled to run on any computing device added to the cross-platform operating environment. A corresponding application environment is automatically created and customized to run the application by evaluating the application requirements, personal settings and platform capabilities. The autonomic operation will free application developers, application users and platform administrators to manually manage the environment settings for individual applications and computing devices. As a result, the ACOE can be used in an open and unmanaged environment such as the Internet to support ODIC and the related on-demand facets, outlined in Section 2.4.

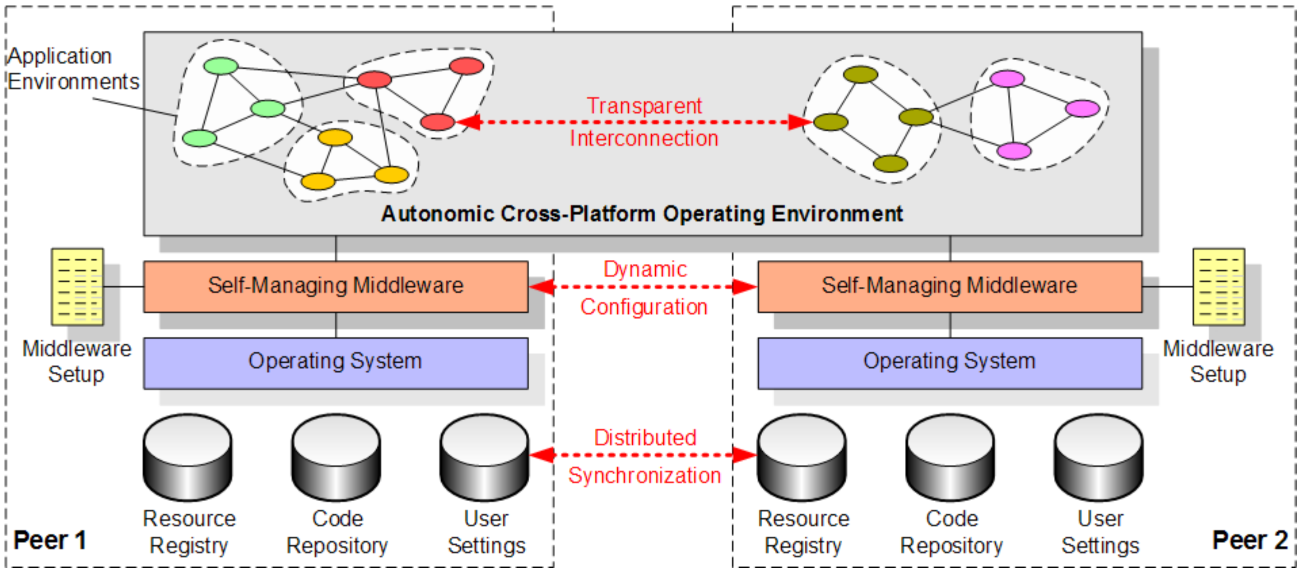


Figure 3.2: Autonomic Cross-Platform Operating Environment

3.1.3 Objectives

In conclusion to the automation of the computing assets and the administration tasks associated with the user roles elaborated in Section 2.3.2 and Section 2.3.3, the following implementation objectives are defined as essential for the ACOE to support the vision of ODIC.

Distributed Code Deployment. In a limited network environment, such as an Intranet, the assembly deployer may distribute all software packages that can ever be requested on every node, but in an Internet environment this is practically impossible. Consequently, mobile code and hot deployment of applications are essential objects. The distribution is performed not only by a single application repository but by various separately maintained repositories. In addition, due to the heterogeneous nature of the Internet, the deployment process cannot be individually prepared for each node but it must be designed to be self-manageable without any user intervention.

Dynamic Software Composition. A major goal in an Internet environment is to avoid transmitting unnecessary data and therefore to reduce the required bandwidth to deploy an application. Internet applications are typically composed of smaller parts that can be individually selected, downloaded and plugged into a running application. Thus, the goal is to support the appropriate self-managed selection of the required component according to the requested operation and the current hosting environment. In addition, components already downloaded by other applications should be stored and shared in a local cache.

Shared Application Hosting. An application developer cannot address every kind of platform architecture or operating system on which the application will probably be hosted in an Internet environment. In turn, a platform administrator is not able to prepare an application system that can host any application, e.g. by installing all existing libraries. Instead, a self-managing application system should dynamically prepare an appropriate runtime environment according to the application requirements and platform capabilities. Another related issue is the provision of a common interface to interact with the platform and its resources, e.g. how to request components or establish network connections.

Nomadic Environment Customization. Users typically customize employed applications according to their personal needs and likings, e.g. picking a certain web proxy server or choosing a particular color scheme. Concerning a cross-platform application environment, an important requirement is the seamless synchronization and application of customized profile settings across the employed platforms without explicit application or user intervention. In particular, this is a non-trivial task in a heterogeneous environment where profile settings may not be directly transferable, e.g. a configured web proxy server could be inaccessible from a different platform and a different proxy has to be automatically determined.

Virtual Object Interconnection. For distributed environments like the Internet a further major issue is the collaboration with remote applications and services. An application can either request remote services and objects or provide access to its business logic, e.g. publishing a web service. However, in heterogeneous environments with changing groups and roles of requestors and providers, there is not always only a single but there are often multiple ways for establishing a connection. Consequently, there is a necessity to dynamically select and create an appropriate network link suitable for the current scenario.

Ad Hoc Execution Migration. The movement of nomadic users requires the migration of running applications from one host to another. The same is valid for remote applications that have to switch the hosting server while they are executed due to load balancing issues or user request. A related requirement is the seamless migration and restoration of the application session as well as the transparent reconnection of remote resources used by the migrating application. In turn, the reestablishment of network links bound to resources of the migrating application is a particular challenge since distributed applications are involved that should actually not be aware of the migration.

3.2 Cross-Platform Operation

A basic objective of a cross-platform operating environment is to enable the seamless execution of user applications across heterogeneous computing nodes. This is typically achieved by introducing a virtual computing layer that provides a uniform computing environment to the application while mediating the interaction with the actual platform installation, e.g. by translating application-level library calls into native system calls. In this section, two major approaches of virtual computing are regarded and the proposal of a *cross-platform operating environment* is deduced for realizing the visions of ODIC as illustrated in Section 2.4.3.

3.2.1 Virtual Machine

A popular approach to enable the cross-platform application execution on heterogeneous computing nodes is the use of a *virtual machine* [385]. A platform-specific variant is deployed by the system administrator on the target computing device where it provides a uniform computing environment by separating the actual application execution from platform-specific characteristics. Based on the abstraction level, related solutions are separated into two major categories, *system virtual machine* and *process virtual machine* [327], as shown in Figure 3.3.

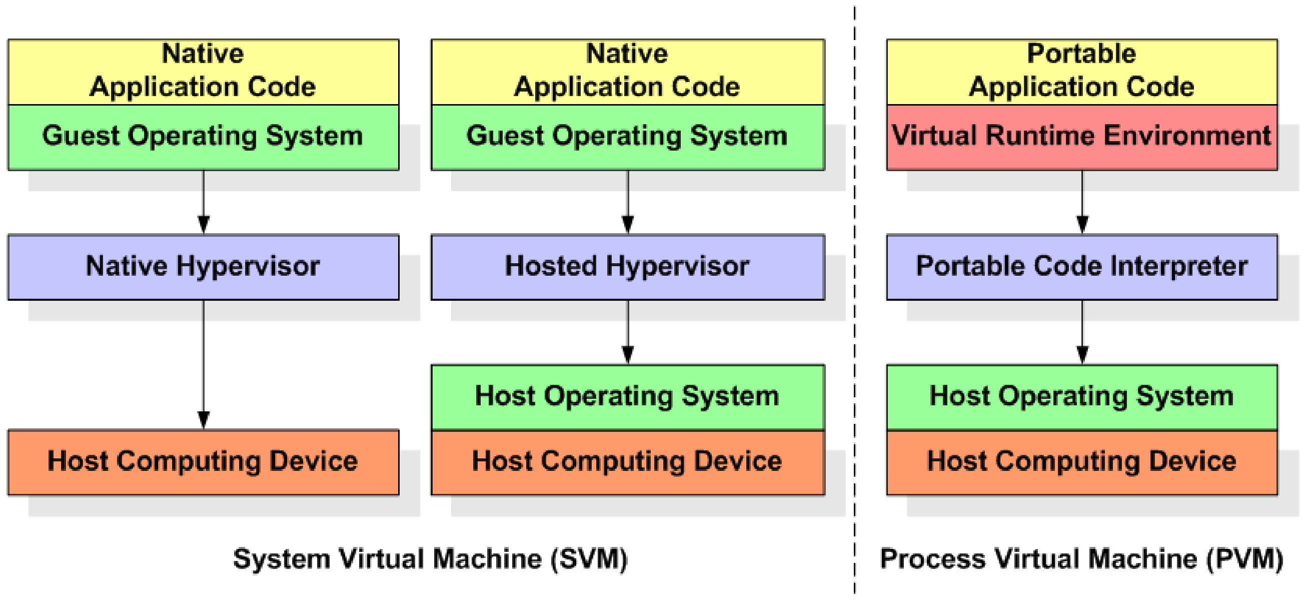


Figure 3.3: Overview of Virtual Computing Approaches

System Virtual Machine (SVM). It provides a virtualized hardware environment that can be almost used like a regular computing device, e.g. for installing a guest operating system and running multiple native applications inside. The main element is the so called *hypervisor* that manages the platform virtualization and allows the concurrent hosting of various guest operating systems. There are two basic types of hypervisor implementations. The native hypervisor (Figure 3.3, left) runs directly on the hardware whereas the hosted hypervisor runs on top of a regular operating system (figure 3.3, center). A native hypervisor is typically used with computing devices that are exclusively dedicated to run virtualized guest operating systems, e.g. virtual servers running in a server environment of an application service provider. A hosted hypervisor is better suited for temporarily launching a virtual machine while the host computer is still regularly used, e.g. for development and testing purposes. Popular implementations of the system virtual machine approach are *VMware Workstation*, *Sun VirtualBox*, *MS VirtualPC* and *Xen* [26] that mainly differ in the handling of privileged native guest code and the exploitation of hardware-supported virtualization features.

Process Virtual Machine (PVM). It runs as a regular program in a host operating system and provides a high-level virtual runtime environment to execute particular applications written in a portable code, also known as intermediate code. A platform-specific *portable code interpreter* (Figure 3.3, right) executes the instructions step-by-step or may use a Just-In-Time compiler (JIT) to compile some parts or the entire application into native machine code before the application is actually started. Advanced JIT implementations operate on a per need basis to avoid delayed application startup, e.g. by analyzing frequently executed code fragments during runtime and compiling only involved code components while the rest of the code is still interpreted. This procedure speeds up the overall application execution with the result that the execution of intermediate code is almost as fast as native program code. Popular implementations of the process virtual machine approach are *Sun Java*, *MS .NET framework* and *Novell Mono* [257].

While both virtual machine approaches, SVM and PVM, may be used to build a cross-platform operating environment on top, they differ in some essential implementation issues. A SVM has to boot the guest operating system first, so its startup time till a specific application can be launched is basically longer than of a PVM that starts the application execution almost instantaneously. Due to that, there are approaches to take a live snapshot of a running SVM and to simply restore it later which is much faster than to boot the guest OS from scratch. The snapshots can be stored in a central snapshot repository and may be deployed to different computing devices, e.g. for setting up spatial computers in an Intranet. A related extension is to maintain so called *appliances*, snapshots with selected applications pre-installed, and to update distributed system installations from a central administration site on a regular basis, e.g. office desktop computers running in an enterprise network or cluster nodes within a Grid installation. Though various appliances may be offered to match the customer's need, only one appliance can be hosted at the same time and a second SVM has to be started in case another appliance is needed. A PVM, in contrast, does not support the appliance approach and there is typically no way to take a snapshot of a running PVM, e.g. for hibernating purposes. This also prevents the spontaneous migration of a running PVM to another computing system while a SVM can be easily moved from one node to another without shutting down the guest applications, e.g. to exploit the computing resources in an application cluster. Due to the *low-level virtualization of a regular hardware environment*, a guest operating system and native applications can be directly run inside a SVM and are usually not aware of doing so. In general, there is no intended interaction between the application and the hypervisor or between the application and the actual hardware components. Various applications may be concurrently hosted within the same SVM while the hosting system and the applications themselves cannot be affected by any application action that is especially important for security sensitive scenarios. A PVM, however, is designed with the goal to provide a *high-level virtualization of a common application environment* and the explicit development of portable applications that are aware of being run by a PVM. As a result, applications can be designed to use common programming interfaces to interact with the underlying hardware, the host operating system and concurrently executed PVM instances. As an example, the invention of *Remote Method Invocation (RMI)* as inherent part of the JVM radically simplified the cross-platform communication for Java applications. The application developer has no longer to deal with low-level network programming, data marshalling or even object serialization. In this context, the application developers also benefit from the portable code and managed execution approach. The same code assembly may be reused and deployed to any PVM installation without considering the target host system configuration. Advanced runtime features provide control over the program execution, e.g. by introducing a code verifier that prevents the loading of malicious components. In comparison to a SVM, the set up of a PVM represents a less time- and resource-consuming task, e.g. multiple PVM instances can be launched from the same installation and share common resources like a local assembly cache.

In summary, the use of a virtual machine eases the deployment and execution of applications without the need to consider varying host system configurations. While a SVM provides a low-level virtualization of a regular hardware environment and executes native machine code, a PVM introduces a high-level virtualization of a common application environment and runs intermediate code. From this point of view, both SVM and PVM may be used to develop a cross-platform operating environment. Concerning on-demand task processing, however, a PVM is better suited to dynamically deploy, compose and run task-specific applications. In addition, the development of particular cross-platform application features like transparent network communication is supported by design.

3.2.2 Features

As deduced in the previous section, a PVM provides high-level virtualization of a common application environment. The major features supporting the development of a cross-platform operating environment are as summarized below.

Cross-Platform Software Development. The invention of a uniform programming environment across different computing systems and the provision of high-level system libraries, e.g. for network, database and GUI programming, allows developers to focus on the business logic without having the need to struggle with different tools and library variants for every target platform.

Simplified Assembly Deployment. The intermediate code representation simplifies the dynamic application deployment across distinct computing systems, especially in a large network environment such as the Internet. The same application assembly may be instantly executed on even yet unknown computing devices given that a suitable virtual runtime system is installed before.

Binary Component Interoperability. A software component compiled into intermediate code can be still inspected (*introspection*). This allows developers to include third-party code components in their applications without having access to the related source code. In addition, application objects can be transferred along with their code, e.g. to enable migration in a cluster computing scenario.

Managed Application Execution. Another feature is the managed execution of the application by using a portable code interpreter approach and monitoring the access to system resources. In fact, a *sandbox* is created from which the application is not able to break out. This is an important feature concerning the dynamic deployment of unknown and potential unsecure code components.

Object-Level Communication. In contrast to native low-level system libraries, virtual machines like the Sun JVM or MS CLR is aware of object entities. If the same virtual machine is used, this enables object-level communication between heterogeneous nodes without considering low-level network aspects. Besides transparent object serialization, a well-known approach is to encapsulate the actual network communication and to let remote method calls do not differ from local ones.

Uniform User Interface. The high-level virtualization of a common application environment also includes the GUI. Platform-specific system libraries provide access to the underlying window system while a platform-independent GUI toolkit can be used to create the illusion of a uniform cross-platform user interface. This also eases the development task since the GUI developed on one computing system is presented just the same on any other computing system without manual adaptation.

3.2.3 Cross-Platform Operating Environment

While a virtual machine already supports cross-platform application execution, the actual operating environment is still limited to a single computing device only. This results in platform-specific configurations that limit the spontaneous employment of a computing device to run yet unknown task applications, e.g. due to different code repositories and task processing policies. A basic idea of this thesis is the introduction of a *cross-platform operating environment* that links distributed computing system to run applications across various computing devices without task-specific preparation in advance. An application framework synchronizes platform, application and user settings with other framework instances to bridge platform boundaries, as shown in Figure 3.4.

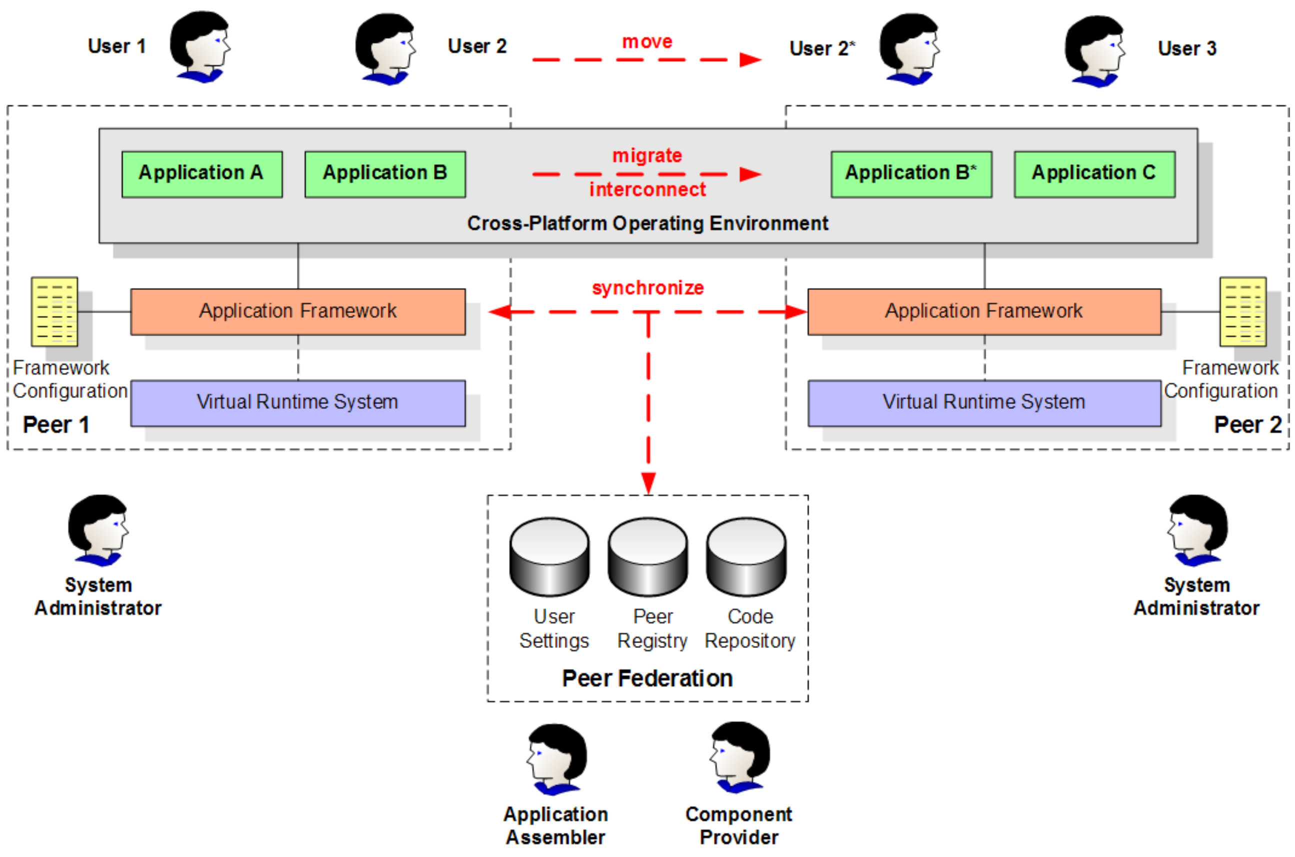


Figure 3.4: Cross-Platform Operating Environment

The application framework is installed and configured by the system administrator on top of a virtual runtime system. It features the resources, capabilities and settings of the platform and enables other peers to access them in a transparent fashion. From this point of view, the assets of distinct computing systems are no longer bound to a single application environment but may be provided by request to any peer within the federation. They may be centrally managed, e.g. using a single registry server, or dynamically discovered, e.g. using a P2P approach. Accordingly, computing tasks may be distributed and run in a cross-platform operating environment on-demand.

In an advanced scenario, the framework also provides a user interface that allows starting a personal session and launching applications on-demand. To this end, the user settings are retrieved from the peer federation, e.g. loaded from a remote peer running a user database, and applied to customize the application environment. The required software components to launch an application are also managed and dynamically exchanged via the peer federation on request, breaking up the fixed deployment of software assemblies into a single code repository. While a computing device may be locally used by a single user only, a remote device may concurrently run various services associated with different users. In this scenario, applications and services may be hosted in the same application environment to benefit from commonly used resources or, alternatively, launched in different application environments without directly interfering with each other.

As a result, users may switch computing devices and do not have to bother with profile synchronization, application configuration and environment customization. In turn, the system administrators may focus on the system setup of a single computing device and do not have to consider which users and applications may utilize the computing device. Application assemblers and component providers can deploy the software packages to any peer in the cross-platform federation and let the peers exchange assemblies directly without using a central code repository. This particularly supports the *"once deployed, run anywhere"* scheme proposed in the ODIC approach. Finally, the role of an application deployer is actually no longer needed since applications are installed and configured by the application framework on-demand the first time they are requested.

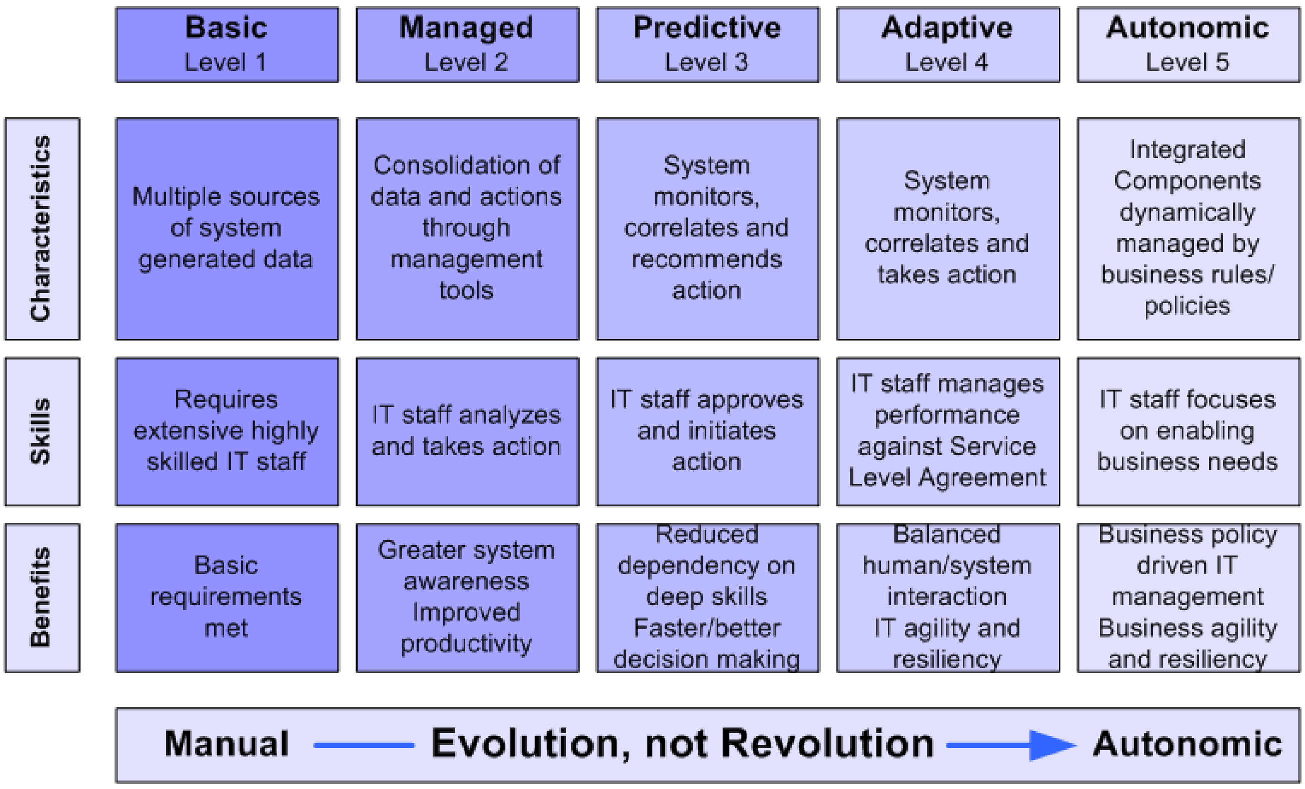
3.3 Self-Managing Operation

As elaborated in Section 3.2, there are various cross-platform issues to deal with for supporting cross-platform computing. An outcome was the proposal of a cross-platform operating environment that is based on an application framework to network distinct computing systems and transparently bridge platform boundaries. In this scenario, it is assumed that the framework is capable of providing and requesting assets without user intervention. However, this is only valid for assets known by the system administrator at the time the framework was installed, e.g. already deployed code assemblies. Recalling the challenges illustrated in Section 2.2.3, the uncertainty of application requests, spontaneous use of heterogeneous resources and dynamic application of user profiles add another level of complexity that cannot be addressed by manual approaches or by one-for-all configurations. A promising solution is the transformation of computing resources into self-managing elements that monitor their environment and react on changing conditions without user intervention, as proposed by the autonomic computing vision illustrated below.

3.3.1 Autonomic Computing

The motivation for autonomic computing stems from realizing that systems have become so complex that they exceed human ability to manage and secure their operation. Major reasons for the increasing complexity are the heterogeneousity and extent of nowadays computing infrastructures [130]. A remedy is their simplification and the provision of a system view that human can understand and cope with [191]. The challenge is to transform computing systems into *self-managing elements* that are capable of quickly responding to requests, changes and failures without or with only minimal user intervention [192]. A driving force behind related *autonomic computing* is IBM which propagates the evolution of legacy systems towards autonomic operation passing various levels of maturity, as described in detail in [250] and sketched in Figure 3.5.

On the *basic level*, the computing system generates complex data from multiple sources that have to be collected and evaluated by highly skilled staff to control the system, e.g. by concurrently inspecting multiple log files. On the *managed level*, management tools ease the access on system parameters via a uniform interface. The staff can focus on analyzing the system and does not have to deal with low-level handling of different log files and configuration setups. Computing systems operating on the *predictive level* support the staff by providing recommendations and preparing corresponding actions. The administrator does not have to understand in detail how the system works but can make decisions upon high-level suggestions.



	Basic Level 1	Managed Level 2	Predictive Level 3	Adaptive Level 4	Autonomic Level 5
Characteristics	Multiple sources of system generated data	Consolidation of data and actions through management tools	System monitors, correlates and recommends action	System monitors, correlates and takes action	Integrated Components dynamically managed by business rules/ policies
Skills	Requires extensive highly skilled IT staff	IT staff analyzes and takes action	IT staff approves and initiates action	IT staff manages performance against Service Level Agreement	IT staff focuses on enabling business needs
Benefits	Basic requirements met	Greater system awareness Improved productivity	Reduced dependency on deep skills Faster/better decision making	Balanced human/system interaction IT agility and resiliency	Business policy driven IT management Business agility and resiliency

Manual —— Evolution, not Revolution ——▶ Autonomic

Figure 3.5: Levels of Autonomic Computing Maturity [250]

On the *adaptive level*, a computing system performs appropriate actions on its own and the administrator is no longer needed to ensure proper operation. The system maintains the basic functionality while the administrator may adjust selected configuration details to meet pre-defined service level agreements. Finally, computing systems on the *autonomic level* operate without human interaction, are able to recover from errors and continuously optimize the task processing by evaluating business rules and policies. The IT staff can handle business requests much faster and does not have to control the actual operation of the computing system. As a result, autonomic computing systems can be especially used to build on-demand computing systems that have to dynamically reconfigure itself to process yet unknown tasks and requests.

While various proposals to build self-managing computing systems have emerged over time [248], the actual autonomic computing approach has been described and issued by IBM in 2001 [169]. It is based on the decomposition of the computing system into autonomic elements that monitor and control the operation of associated resources [381], as shown in Figure 3.6.

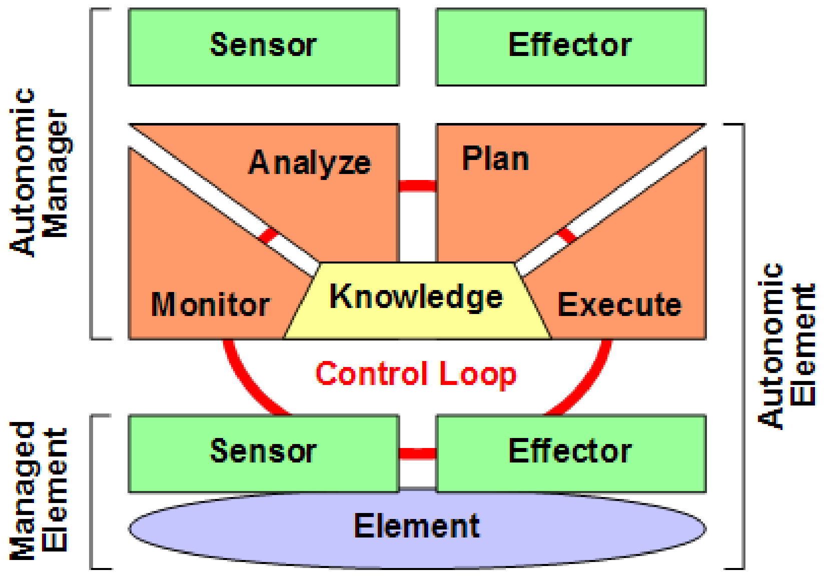


Figure 3.6: Autonomic Manager and Control Loop [250]

The managed element can be any resource in an autonomic computing system, such as a software module, a network connection or a database. It features a sensor to retrieve information about the condition of the element, e.g. by registering an event trigger, and an effector to adjust the resource behavior and state, e.g. by using an Application Programming Interface (API) to issue commands to the resource. The autonomic manager utilizes the sensor and effector to build a closed control loop that monitors the resource condition, analyzes the current scenario, creates a plan to maintain the supposed function and executes necessary changes on the resource. Besides predefined system policies that outline the basic operation of the control loop, the autonomic manager ever improves its knowledge by learning about the managed element, e.g. by tracking of instantiated software components and reusing compatible ones instead of loading another variant. Learning can be used to predict future situations and create an adaptive response to yet unknown challenges. As a final result, the autonomic manager transforms the managed element into an autonomic element that is capable of responding to resource issues in a self-managing way and at best without user intervention.

3.3.2 Features

The introduction of an autonomic manager enables the realization of various self-managing features, subsumed under the term *Self-X* [169]. Besides the original ones proposed by IBM, namely self-configuration, self-optimization, self-healing and self-protection, further features have been proposed by related approaches like self-organization and self-explaining in *organic computing* [317]. Below, self-managing features are listed with particular respect to the anticipated features of an autonomic cross-platform operating environment to run Internet applications on-demand.

Self-Description. An autonomic manager is responsible for handling requests received by the application and for controlling the managed element, e.g. a platform resource or remote service. In a distributed and heterogeneous environment like the Internet, the elements to be managed are not known in advance and may change across different platform installations. Nevertheless, the application framework should be able to adjust its operation without user intervention. Remedies are self-descriptive elements that describe their capabilities and requirements in a standardized way, e.g. by XML formatted configuration files. From this point of view, self-description is an essential feature of an autonomic cross-platform operating environment.

Self-Configuration. The second feature is the ability to prepare the target node according to the requirements of the application and with respect to the capability of the currently involved computing system. Furthermore, migrating applications and nomadic users come with specific profiles about their global and platform-specific preferences that may have to be moved and processed as well. The interconnection with remote objects cannot always be configured in advance, but current network constraints such as blocking firewalls must be considered. As a result, self-configuration enables applications to be dynamically requested on a different Internet application system without intervention by the user or the application, respectively.

Self-Integration. To deploy an application, already installed components and available features of the currently used Internet application environment must be considered. The application should seamlessly integrate itself into an application environment and utilize existing features instead of asking for a separate installation. To this end, the application system should be customized according to the application requirements and platform capabilities. As an example, instead of downloading application specific components for each application separately, already installed compatible components should be reused. Similarly, an existing network link to a remote object can be shared among applications instead of establishing further communication paths.

Self-Optimization. Once a cross-platform application has been distributed and a suitable configuration has been determined, there is still the need to survey the existing configuration whether the operation can be improved to better fit to the current application requirements and system conditions. For instance, the release of new software libraries could offer the update of already deployed and configured applications. Running applications may have to migrate to another computing system, and established interconnections may be re-routed due to changing workload. Therefore, autonomic cross-platform computing should provide facilities to monitor existing application installations in a self-optimizing manner to make them more efficient.

Self-Healing. In contrast to self-configuration and self-optimization, which are derived from given high-level objectives with respect to expected conditions of an application, there also is the need to deal with unexpected conditions such as software failures, network interruptions or runtime errors. In this sense, self-healing is the ability to repair parts of an operational system without shutting it down. As an example, a remote object connection might be lost due to the migration of the object to another host and must be individually re-established. In another scenario, an application may use a software component that suddenly fails and which then has to be dynamically exchanged. Thus, self-healing is another objective that should be supported by an autonomic cross-platform computing environment.

Self-Protection. Another objective is the prevention of errors arising from security related problems that cannot be managed by self-configuration or self-healing measures. To this end, self-protection essentially organizes the defense against malicious attacks or incorrect usage. For distributed Internet application scenarios, this objective is important to ensure proper operation of applications. For instance, network communications should be appropriately encrypted, if necessary. Deployed software packages have to be validated before they are used, and the distribution of user profiles should only be allowed to trusted platforms.

3.3.3 Self-Managing Infrastructure

Besides the cross-platform features that are related to the functional operation of the cross-platform operating environment, it is interesting to introduce cross-cutting autonomic features that deal with the timely provision of Internet computing assets in a self-managing way as presented in Section 2.3.2 and Section 3.3.2, respectively. Transferring the idea of autonomic computing to the cross-platform operating environment presented in Section 3.2.2 to realize a self-managing infrastructure, the platform resources are identified as the managed elements, the application framework as the autonomic element and the cross-platform application as the requesting element, as shown in Figure 3.7.

The platform resources at the bottom represent the assets provided by the computing device. They are configured by the system administrator and likely differ from one computing system to another. In turn, the configuration of the cross-platform application has been defined by the application developer and will typically not change across distinct platforms. The application framework is the autonomic element sitting in between.

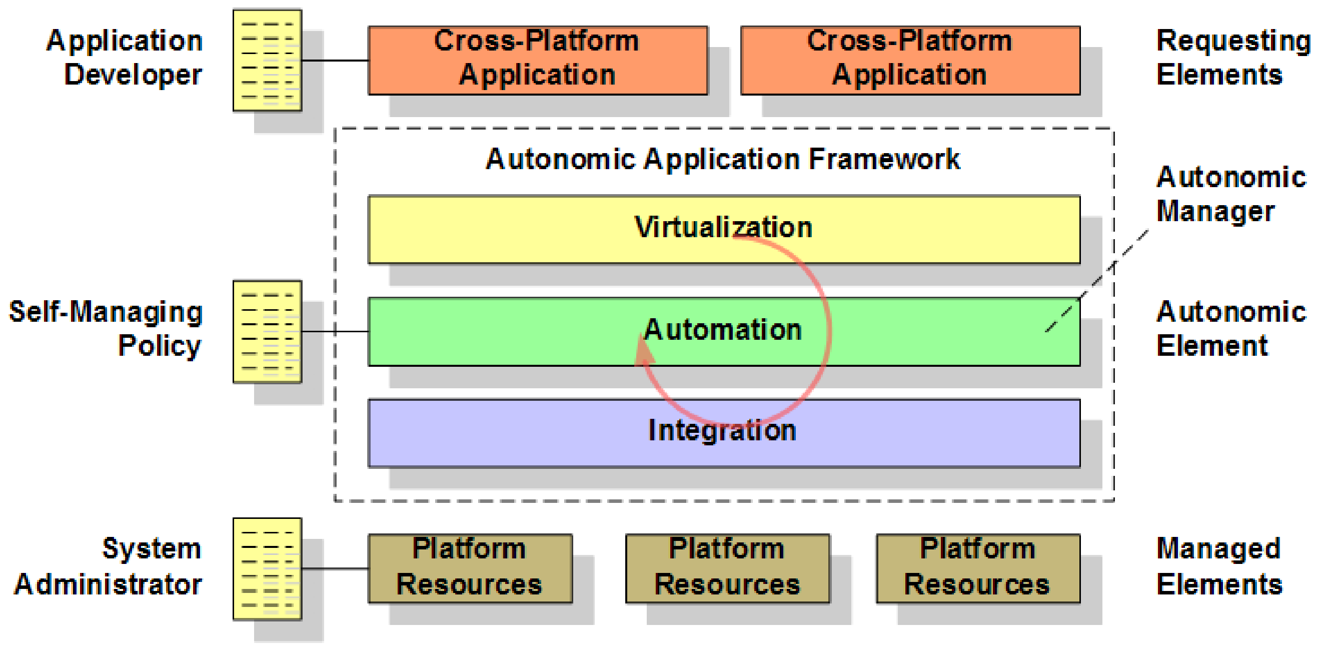


Figure 3.7: Self-Managing Infrastructure

The particular challenge is to add cross-cutting autonomic behavior to the application framework to enable autonomic operation without changing the concerned applications and resource on the one hand. On the other hand, the automation should be separated from the concrete platform configuration, and the application should be shielded from the automation details. To this end, the autonomic application framework is separated into three layers. The *integration layer* provides uniform access to platform resources across heterogeneous computing devices using an integration middleware. This eases the implementation of the autonomic manager residing in the *automation layer* that does not have to consider different platform capabilities and configurations. The autonomic manager performs the actual self-managing tasks and is used by the *virtualization layer* for providing a cross-platform operating environment to the applications without revealing automation issues to the applications.

As a result, the self-managing infrastructure replaces the manually performed tasks of the user roles concerning asset provision described in Section 2.3.3 into *self-managing operations* dynamically performed by the application framework, as proposed in Section 3.3.2. Particular intervention is no longer needed to prepare a suitable application environment but the system itself handles user and application requests. The self-managing infrastructure is also able to react to changing runtime conditions and adjust the configuration of the cross-platform operating environment.

3.4 Supporting Solutions

In this section, potential solutions and their applicability to support autonomic cross-platform operation, as outlined in Section 3.2 and Section 3.3, are discussed. The solutions are categorized along the dimension of the cross-platform operating environment they are supposed to support, namely a single computing device, the Intranet, the Extranet and the Internet, as introduced in Section 2.2.1. Consequently, *single computing*, *enterprise computing*, *community computing* and *public computing* are identified, as shown in Figure 3.8 and detailed below.

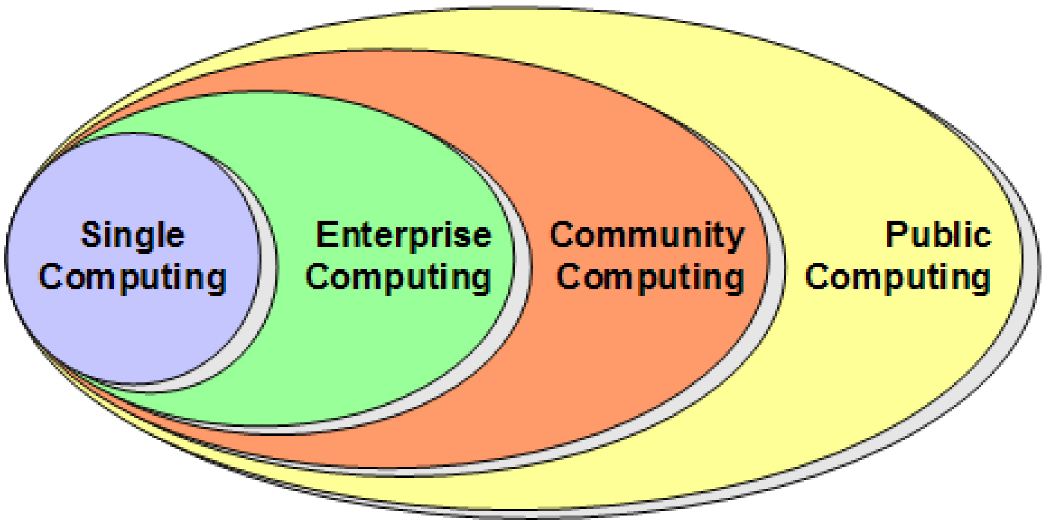


Figure 3.8: Types of Cross-Platform Operating Environments

3.4.1 Single Computing

The operating scenario *single computing* is related to an operating environment limited to a *single computing device* only, typically managed by an administrator or a regular user. Although runtime operations are not synchronized with other computing systems, single computing solutions introduce various cross-platform features to ease the software deployment to and the employment of heterogeneous computing devices. The administrator installs an operating environment customized to the platform capabilities to free application developers from dealing with hardware concerns and low-level system operations, e.g. by using native operating systems, virtual machines and particular application frameworks.

Native Operating Systems. With the release of Windows XP, Microsoft has introduced some autonomic features with respect to self-configuration, self-customization and self-healing. The software and hardware installation is simplified by various installation wizards, user and hardware profiles ease switching to another computing system, automatic updates ensure the installation of latest patches and the system restore feature protects core system settings by supervising installation tasks. Concerning cross-platform support, most of the features of MS Windows XP may be only used in a pure Windows environment, e.g. using image files to redistribute a Windows installation on equally configured computing devices. There is no or only little support to use Windows-specific tools and features in another operating system, e.g. applying Windows user profiles on Linux systems. Similar to MS Windows XP, Linux represents an operating system that has to be manually installed and configured on the target computing device. However, there are various Linux distributions a system administrator may choose from which in practice support almost any computing architecture from large-scale server systems down to smart phones. As a consequence of this diversity, Linux binaries cannot easily be transferred from one Linux system to another. This complicates software deployment compared to MS Windows. Software package managers, such as RPM or APT, offer automated installations of suitable software packages using self-configuration and self-customization features. Linux also enables system switching and user profile synchronization using directory services, such as LDAP [205]. In comparison to MS Windows, the self-managing features of Linux are still in an early development stage and allow only experienced users to master Linux on desktop computing systems.

Process Virtual Machines. While native operating systems can run only platform specific program binaries, a virtual machine can host related applications independent of the actual computing device. As deduced in Section 3.2.1, a process virtual machine (PVM) fits better into an on-demand application scenario than a system virtual machine (SVM). Currently, there are two main PVM solutions with a widespread installation basis, Sun Java and Microsoft .NET [108]. While there are many discussions and arguments about their individual merits, both solutions co-exist in different application scenarios. In fact, Sun Java is widely used in the Internet composed of heterogeneous computing devices, most of them running MS Windows and Linux. Microsoft .NET rather dominate in a homogeneous environment running MS Windows, e.g. in a corporate Intranet.

Sun Java benefits from a virtual runtime system that enables Java applications to instantly run on any computing device with a Java Runtime Environment (JRE) installed. It consists of a platform-specific Java Virtual Machine (JVM) and corresponding runtime libraries that enable the seamless execution of Java applications on a variety of platforms [68]. A Java compiler generates the intermediate program code, called Java byte code, which may be used to compile further applications without accessing the original source code. This facilitates cross-platform operation like software deployment and application hosting in a heterogeneous environment such as the Internet, e.g. Java applets may be deployed and executed with any Java-enabled web browser. Built-in support for cross-platform network communication based on CORBA and Web services ease the transparent interconnection of Java applications with various remote computing systems. Since a JRE is available for virtually any computing device, from mobile phone to a server system, Java is widely acknowledged to be the first choice for running applications on heterogeneous platforms. Concerning autonomic operation, *Java Web Start* offers self-configuring deployment and runtime provision for legacy Java applications. In contrast to MS Windows and GNU Linux, however, a regular JRE does not support the synchronization of user and runtime profile across various platforms and application installations, e.g. if a user moves to another computer system. Another limitation is the original design of the JRE to run a single application only. A common remedy to easily run multiple Java applications within the same JVM is the initialization of a Java launcher. It is a regular Java application that loads and launches further Java applications, e.g. by offering an interactive command line shell, such as *Jsh* [182], or an application menu, such as *Xito AppManager* [390]. Moreover, there are solutions that attempt to create a Java-based operating system and an extended Java runtime system, such as *JOS* [181], *JNode* [180] and *JX* [138, 184]. The common idea is to provide a bootable image written in native code that is directly run by the computer system instead on top of a commodity operating system like MS Windows or Linux. Apart from concurrently launching various Java applications, there are various arguments pro Java OS, e.g. increased overall performance and better system integration, e.g. by addressing low-level system support. However, this approach represents a most complex development task. In fact, no Java OS has ever left the beta development stage and has been widely adopted on desktop and server computer systems. Concerning the support of on-demand task processing in a cross-platform operating environment, a basic issue is the dedicated use of the target computer system and the installation of an unmanaged runtime environment in contrast to run a single or multiple legacy JVMs.

Although the regular JVM is executed on a single computing device only and represents a multi-platform runtime system, the JRE is often called a cross-platform runtime system. The term cross-platform refers to its inherent ability to support the distributed execution and fragmentation of an application across heterogeneous computing devices, e.g. using Java RMI and dynamic byte code transfer. From this point of view, the virtual runtime system .NET from Microsoft follows a different goal [237]. Instead of providing specific virtual runtime systems for different computing platforms, Microsoft has developed the .NET Common Language Runtime (CLR) for tight integration into its operating system Microsoft Windows only. The focus is on the support of executing intermediate code built from different programming languages, such as .NET C++, .NET C# and .NET Visual Basic. Consequently, the original .NET framework can only be used for cross-platform computing based on MS Windows. Nevertheless, it provides a managed runtime environment that offers self-managing features, such as self-configuring software composition using assembly metadata. The drawback of .NET concerning the support of host systems other than Microsoft Windows is addressed by the open source project *Mono* that aims to port the .NET runtime system and framework on heterogeneous host systems, such as Linux, Mac OS and Sun Solaris. However, Microsoft does not officially support the development of Mono and attempts to restrict essential parts of the .NET framework, such as ASP.NET and *Extensible Application Markup Language (XAML)*. Other parts are released to the *European Computer's Manufacturer Association (ECMA)* for standardization, such as the programming language C# [253]. This hinders the complete clone of the .NET framework and so far has also prevented the use of .NET in a cross-platform manner as demonstrated with Sun Java.

Rich Client Platforms. In a typical single computing scenario, the administrator of a computer system installs distinct software applications in parallel. Apart from system libraries, there is typically no reuse of common code assemblies or application configurations. Further, applications are executed in different process environments and do not share runtime resources, e.g. an application window or plugin repository. A popular remedy is to introduce a *Rich Client Platform* that offers common services to the applications, e.g. unified configuration, window management and plugin management. It is installed on a desktop computer in advance and provides a shared installation environment to distinct applications and plugins. For example, *Xito AppManager* defines a rich client application model to run multiple Java applications in the same JVM [390] and provides cross-application services to organize the handling of shared resources. It also offers advanced options to retrieve software components and application configurations from well-known remote software repositories, e.g. via the Internet. As a result, an application developer can focus on the actual business logic and users benefit from an integrated application environment. A basic problem of custom application launchers is their proprietary implementation and lack of developer support. In this context, the *Spring Framework* represents a widely adopted application framework that simplifies the development of Java applications by providing a set of common libraries and services, e.g. for component initialization and lookup. Based on the Spring Framework, the *Spring Rich Client* [330] provides a Rich Client Platform for Java desktop applications and aims at providing a unified application environment and runtime model. In this context, the *Open Services Gateway Initiative (OSGI)* specifies a Java-based application framework targeting a wide range of computing devices like desktop computers, small computing devices and embedded systems [262]. It provides a *"service-oriented, component-based environment and offers standardized ways to manage the* *software lifecycle"*. The framework benefits from the advantages of Java as described above and it adds further cross-platform features concerning software deployment, application composition and runtime management. The introduction of so called *bundles* allows developers to define interdependencies between software components and enables the self-managing composition of Java applications from a shared bundles pool. As an example, *Exymen* utilizes an OSGI framework to manage and update application plugins [105]. A user can easily retrieve and install new plugin bundles, e.g. a media editor plugin, from dedicated bundle repositories without being involved in resolving plugin dependencies. By extending the features of the regular Java Virtual Machine (JVM), OSGI also supports multi-application hosting and separately configurable application environments within the same JVM. Popular implementations of the OSGI specification are *Apache Felix* and *Eclipse Equinox* that are supposed to run on desktop computing devices and represent Rich Client Platforms [13, 95]. Basically, Eclipse Equinox represents a good choice for running Java-based OSGI services and deploying rich client applications via the Internet. It is used in various Eclipse subprojects, such as *Eclipse Rich Client Platform (RCP)* and *Eclipse Java Development Tools (JDT)* [96, 154]. There is a strong developer community, and a growing number of OSGI-applications can be run in Equinox out of the box. The original OSGI framework, however, focuses on the support of single computing scenarios by managing the deployment and execution on distinct computing nodes separately. There is no support for synchronizing user and application profiles across various installations, and the core components of Equinox have to be manually installed by an administrator. Further, the on-demand resolution and installation of missing bundles from remote software repositories is typically not part of the OSGI engine implementation. Thus, it may launch only applications that are already known at the time the on-demand processing request is received. In addition, the OSGI specification introduces a new programming and application model that forces developers to rewrite legacy Java software components.

As a result, there are various native operating systems, such as *Apple OS X* and *Sun Solaris*, virtual runtime systems, such as *Java*, *Perl* and *Python*, and Rich Client Platforms, such as *Spring Rich Client* and *Eclipse Equinox*, which address the single computing scenario. They differ in terms of the supported hardware platforms, programming language and application model but commonly miss some cross-platform features and the autonomic operation as proposed for the ACOE. In summary, the regarded approaches focus on operating scenarios managing a single computing system with static hardware constellations and well-known application tasks. There is no regular support to synchronize the deployment, composition and customization of applications across multiple computing systems. From this point of view, the single computing approaches are not suitable to create an autonomic cross-platform operating environment with distinct computing systems involved, but they may serve as a foundation for advanced approaches, as described in the next sections.

3.4.2 Enterprise Computing

Another operating scenario is *enterprise computing* with an operating environment spread across the computing devices of a company, also denoted as *Intranet*. The computing systems are managed by an administrator group that assigns every device a specific task within the enterprise network, e.g. hosting a database instance, and prepare a suitable configuration for each one in advance. The resulting cross-platform operating environment organizes the runtime operation of tightly related, often homogeneous computing systems according to the enterprise objectives, e.g. creating a *high-performance computing cluster* [24]. In contrast to the single computing scenario and its box-centric view, multiple computing nodes are interconnected and configuration changes of one node have to be propagated to all nodes as well; typically in a manual fashion. Thus, in a classic setup, the computing nodes are coupled in a static way and employ the same operating system and set of business applications to ease the management. A selection of enterprise computing solutions is described below.

Enterprise Application Servers. The transformation of the web server into an application server has lead to various approaches and standards to ease application development and deployment. A popular approach is *Sun Enterprise Java Beans (EJB)* [336] that has become the de-facto standard for Java application servers and is used in various products, such as *JBoss* or *Jonas* [179, 187]. The major advantage is the introduction of a server-side component model that supports the application composition using third-party components. Instead of implementing every application from scratch, the application is deployed using a deployment descriptor. The application server evaluates the application configuration and prepares a suitable runtime environment for every service. Concerning cross-platform operation, the EJB standard enables the seamless deployment of components across different application servers. An application developer does not have to consider a specific server implementation, and the administrator may choose the right application component out of a set of compatible variants. In a remote task processing setup, application servers can expose certain object instances via a network protocol, such as RMI, CORBA or SOAP. A naming service, such as the *CORBA Naming Service* or the *RMI registry*, is used to register and link remote server objects. In an advanced task processing scenario, a component search engine may be introduced to find a suitable object implementation on-demand, e.g. by specifying a complex component query as in the *AGORA* system [320]. Further, multiple application servers may be grouped to form a distributed task processing platform. A related example is the *Java Parallel Processing Framework (JPPF)* that splits applications in smaller parts and enables the simultaneous task processing on different machines [183]. Another example is *Entropia* that uses MS Windows desktop computers to enable distributed task processing [59]. It hosts native applications in a sandboxed environment and provides a central job management to control the *desktop grid*. The actual installation and configuration of the application server and its processing objects, however, is mainly done in a manual way and with particular consideration of the tasks for which the box is supposed to be used. The application server approach is not intended to be used on the client-side and also does not support user profile management across distinct computing devices.

Remote Application Terminals. Another enterprise computing solution is the use of an application server to centrally manage desktop applications, such as offered by the *X Window System* and *Virtual Network Computing (VNC)* [389, 300]. Instead of deploying all applications on every desktop computing system, each application is installed on a central application server only. A particular application client is installed on the desktop computer that is able to launch an application on the application server and to redirect user input to the application and to receive updates of the user interface. As a result, the user can interact with the remotely executed application as with any regular desktop application. Actually, the application client represents a universal networked user interface that may be used to control different and unknown applications run on the application server. From this point of view, a user is able to instantly run any application as soon as the administrator has installed it on the application server. Apart from maintaining the application client, the administrator does not have to deploy new applications or software updates to each desktop computer system. Concerning cross-platform operation, the application terminal solution enables users to access applications without the need to run the related program code, e.g. Linux clients may use MS Winword. However, the application server becomes a bottleneck if many users concurrently launch applications, and a regular user is not able to run applications not yet installed on the application server. In addition, a fast network connection is needed to get the illusion of running a regular desktop application and offline operation is not supported [211].

Service-Oriented Architecture. The evolution of enterprise networks over time and the resulting diversity of employed computing systems end in separate and heterogeneous computing resources that cannot interact although they are connected to the same Intranet. Instead of simply connecting disparate computing assets, resources have to be integrated and must become part of the overall system. A recent approach is the invention of an *Enterprise Service Bus (ESB)* based on a *Service-Oriented Architecture (SOA)*. Each resource of the enterprise network is transformed into a service that is transparently connected via the ESB. The ESB mediates the interaction among the services using appropriate communication protocols, e.g. *Java Message Service (JMS)* and *Simple Object Access Protocol (SOAP)*. Additional elements like a *Message Queue Server (MQS)* introduce new features, such as service synchronization and transaction control. Actually, the use of an ESB has become a key element of today's enterprise computing solutions. With the dynamic orchestration of web service workflows, e.g. by using the *Business Process Execution Language (BPEL)*, complex web service interactions may be automated. This can be refined with the automated provisioning of application services to enable on-demand task processing [190]. Various products, such as *HP Adaptive Enterprise*, *IBM Tivoli Intelligent Orchestrator* and *IBM On Demand Operating Environment* [51, 231], promises the seamless collaboration and central management of networked resources without replacing matured installations and thus increase the utilization and reduction of maintenance costs of enterprise resources. Concerning cross-platform operation, SOA and ESB solutions focus on the management and collaboration support of already deployed and configured resources and services. As a result, the system administrator still has to perform deployment and configuration tasks of certain services and prepare a suitable runtime environment. The support of user-related issues, such as profile synchronization, is usually not addressed by an ESB implementation.

Ad-hoc Service Infrastructure. While the cross-platform operation support of application server frameworks automates the preparation of an application environment to run already configured and deployed applications, there are enterprise computing solutions supporting the spontaneous access on yet unknown remote services, such as *Universal Plug and Play (UPnP)*, *TSpaces* and *Sun Jini* compared in detail in [156]. For example, Sun Jini uses directory services to announce new services and resources, such as data decoding service and a color printer, and to deploy corresponding network stubs required to access these services and resources. An application looking for certain Jini services queries the directory service for matching service descriptions and downloads the related network stub to access the remote service. As a result, an application may dynamically access discovered services without having been explicitly configured to do so in advance. By using a service interface specification and downloading a specific stub for each service, the implementation of the service and the application are separated, which eases their deployment in a cross-platform operating environment. In addition, the announcement and discovery of new services is performed in a self-managing way, e.g. using leases for service validation, without requiring particular application or user intervention. The self-organization features of Jini may be also used to simplify the management of clustered server systems like in *Large-scale system's Autonomous Management Agent (LAMA)* [219]. In general, an ad-hoc service infrastructure is typically limited to a subnet and a small number of computing devices. By focussing on the goal of providing on-demand access to remote services, there is usually no support of dynamic application deployment and composition; user profile synchronization is also not addressed.

As a result, the common goal of enterprise computing solutions is the support of administrators to manage a limited number of well-known networked resources, such as application servers, desktop computers and peripheral devices. The solutions differ with respect to their management support, e.g. dynamic application composition as with Enterprise Java Beans, service orchestration as with IBM WebSphere and self-managing service discovery as with Sun Jini [374]. There are further enterprise computing solutions, such as *Sun N1*, *HP Adaptive Enterprise*, *MS Distributed Systems* and *IBM On Demand Operating Environment* [51], which address the management of computing resources from different points of view, e.g. data management centers, heterogeneous network elements, distributed computing systems and on-demand service provision, respectively. To summarize, enterprise computing solutions represent a good choice to ease the management of well-known enterprise resources typically found in a managed network environment, such as databases, application servers and web portals. Autonomic features focus on the self-managing operation due to changes imposed by alternating tasks, such as launching a new application on a VNC server or integrating a new EJB component in an application server. However, enterprise computing solutions are limited to manual integration of additional computing devices, e.g. installing the VNC client on new desktop computers, and assume static platform configurations, such as using a well-known relational database with container-managed *Enterprise Java Beans*.

3.4.3 Community Computing

The third operating scenario is *community computing* with a managed cross-platform operating environment spread across heterogeneous computing devices of various organizations and networks that form an *Extranet* [41]. The computing systems are separately set up by different administrators following a common setup guideline for integrating each one into the distributed operating environment. Multiple heterogeneous computing devices are loosely coupled and alternately used to process tasks yet unknown when they are set up. The actual difference to enterprise computing is the wide-area extension and the loose coupling of applications, e.g. by using standard service protocols, such as *Simple Object Access Protocol (SOAP)* and *Agent Communication Language (ACL)*

[120], instead of program-specific elements, such as proprietary service stubs and software components. This also enables the dynamic composition of distinct web services, known as *orchestration*, that have not especially been designed to work together in the first place. A particular extension is to share virtualized computing resources on a pay-per-use basis and to outsource the operation of custom application services to specialized providers, as in utility and cloud computing [372]. Personal and terminal mobility are addressed by mobile code approaches. Selected examples of community computing solutions are discussed below.

Web Services. With the ability to penetrate firewalls using HTTP communication and to call remote services using standardized XML encoded messages, *web services* have become a widely used approach to bridge the Internet. Communication protocols, such as XML-RPC and SOAP, allow applications to access remote services in a platform- and programming language-independent manner and also provide the basis for establishing a general-purpose service infrastructure [58]. A key stone in this picture is the use of particular registry services, such as *Universal Description, Discovery and Integration (UDDI),* to announce the availability and characteristics of new services on the one hand and to query for present and matching services on the other hand. The registry service may be additionally used to exchange service interface descriptions, e.g. specified using *Web Service Description Language (WSDL)*, and to enable clients to create a corresponding service stub on the fly [73]. From this point of view, related solutions, such as the *IBM Web Service Gateway*, support spontaneous cross-platform operation among loosely coupled service providers and service consumers. However, service-oriented approaches typically do not address the deployment and configuration of clients and services but focus on their interaction on the network level. There is no guideline how to set up distinct web service nodes. The installation is often performed in a specific way with respect to the employed application server, e.g. deploying a web archive (WAR) into a servlet engine. Moreover, the underlying communication model forces application developers to alternate between object-oriented implementation features, e.g. native object-serialization in Java, and service-oriented design limitations, e.g. lack of real object references and callback methods. This complicates the mixing of existing application code and new service code.

Service Grids. While web services are typically used to access a specific remote computing system, *service Grids* transparently group distinct computing nodes from various institutional and organizational domains into a virtual computing resource. The Grid allows performing lengthy tasks that are otherwise too complicated for a single computing system to deal with. Various solutions, such as *Globus* [117, 135], *Unicore* [102, 361] and *Condor* [354, 64] support the establishment of the basic

Grid infrastructure by adding particular architectural elements, such as a task scheduler and a resource monitor. The solutions mainly differ in the target application scenario, project partners and underlying standards, such as the *Open Grid Service Architecture (OGSA)* [118, 350] and the *Web Services Resource Framework (WSRF)* [259] A common advantage is the dynamic allocation of computing nodes, the launching and orchestration of required services and the transparent distribution of computational tasks. While Grid approaches originally rely on service installations in advance, dynamic service and job deployment is also supported to utilize Grid nodes for applications yet unknown when the Grid was set up. By using virtual machines, Grid resources can be virtualized and the heterogeneity of the participating computing nodes masked [111]. Advanced approaches may establish virtual distributed environments in a Grid to run different applications in shared and isolated settings [309]. From this point of view, Grid solutions support cross-platform operation in terms of deployment, composition, hosting and interconnection of services. Various autonomic features, such as self-configuration and self-integration, are supported and there are efforts to develop autonomic Grid applications, as pursed in *AutoMate* [3]. However, Grid solutions like Globus address distributed application scenarios on the server side and are not usable on the client side. The cross-platform operation is limited to a managed infrastructure with well-known platform configurations and runtime environments, e.g. there is no need for application developers to consider different platform capabilities and software composition as found in a heterogeneous environment.

Cloud Computing Platform. Due to the ubiquitous availability of an Internet interconnection, the falling costs of computing systems and data storages, and the progress in virtualization technologies [86], the approach to exploit remote computing resources in an on-demand manner has been recently extended towards *Everything-as-a-Service (XaaS)* [220]. The basic principle is to virtualize computing resources like system hardware, network infrastructure, operating system and middleware platforms with the goal to enable their provision and utilization as services via the Internet. Besides private cloud installations that are limited to the use in an enterprise environment, there are public cloud installations, such as *Amazon EC2* [9] and *Google App Engine* [143], which are made available to everyone on a pay-per-use basis. From this point of view, cloud computing is related to utility computing and is typically separated into *Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS).* The resource management is mostly performed in a self-managing and transparent way, e.g. by moving a software service to another computing node in case of a hardware error. As a result, the customer is relieved from administration tasks, however, at the expense of security risks and privacy threads by moving business applications and private data to a remote site [369]. A related solution is *TeamDrive* that allows users to store documents in the *Amazon Cloud* and to transparently access them like regular files from any Internet workstation, e.g. running at the office or at home [353]. Concerning cross-platform operation, a cloud installation delivers a virtualized and scalable operating environment across heterogeneous computing systems. Depending on the concrete cloud installation, the dynamic deployment, composition and hosting of regular Internet applications on-demand is supported with slight differences compared to a native execution environment, e.g. Google App Engine offers the hosting of custom Java applications running with a feature limited JDK only [143]. The transparent interconnection and migration of services is addressed by various implementations like *Eucalyptus*, *Open Nebula*, *Intergrid* and *GridGain* that use cloud technologies to establish a virtualized distributed environment across multiple Internet computing sites [65, 103, 328, 149]. As such, cloud computing is often seen similar to the Grid approach but it does not target the particular issues for controlling multiple nodes to compute a given a task in parallel, as elaborated in [119]. Besides distributed task processing, cloud installations may be also used for remote task processing by using regular desktop computers and following the public and volunteer computing idea, as proposed in *Cloud@Home* [72]. The hosts act as a consumer and provider of computing resources at the same time, thus forming a dynamic computing community, as discussed in [175]. Nevertheless, a basic issue with cloud computing remains the installation and configuration of the resource virtualization and the dedicated use of the computing node. This makes it difficult to use cloud computing solutions for local task processing.

Mobile Code Systems. Another community computing approach is the use of *mobile code* to push applications and services to networked computing nodes. For remote and distributed task processing, an agent network infrastructure [386] allows running autonomous and proactive applications, so called *mobile agents* [294], across distributed computing systems. Each agent pursues a certain goal, may communicate with other agents to request resources or, vice versa, offer own resources to requesting agents. An agent is able to move from one host to another host that are often grouped following the P2P model, such as *AntHill*, *JATLite* and *Aglets* [12, 185, 212, 4]. A representative implementation is *JADE* [178] written in Java, which represents a multi-agent middleware to run agents on various types of hardware, operating systems and Java Virtual Machines, e.g. J2SE and J2ME. It consists of a specific runtime environment for each platform variant, an agent container, which must be manually installed by the system administrator and provides libraries to link the actual agent against. Jade is supported by the *Foundation for Intelligent Physical Agents (FIPA)* and uses the standardized *Agent Communication Language (ACL)* to interact with other agent containers [120]. This ensures interoperability with different vendors and implementations of agent containers.

Concerning cross-platform operation, JADE offers the spontaneous deployment, seamless execution, transparent interconnection and ad hoc migration of agents. A related multi-agent system is *AgentScape* that has been designed for large-scale network environments [46]. It supports multiple code bases and provides a flexible application framework that separates platform dependent and independent parts. In addition, it supports the deployment of agents written in diverse languages, such as Java, Python and C. Though various mobile agent toolkits exist, they are not yet used to implement autonomic application built of mobile agents [48]. In contrast, autonomic elements are rather framed as services than agents. Another drawback of agent networks is their explicit focus on a specific application scenario, namely the collaboration of autonomous applications. There is no or only little support of user interaction, e.g. supporting nomadic users while moving from one computing system to another. The programming model is different compared to regular applications, e.g. it uses a particular action method as execution starting point and is limited to single threading, and there is no support of application composition or user profile synchronization in general. Another mobile code approach is used in the *Thin Servers* solution that supports the dynamic deployment of Java task applications, so called *Cingalets* [85]. It explicitly focuses on remote task processing and uses native C++ daemon processes to startup separate JVMs and run the dynamically deployed Java task applications inside. For local task processing, various personal computing solutions, such as envisioned by *Universal Personal Computing* [207, 222] and implemented by *NetChaser* [333], have been proposed to support *terminal mobility* and *personal mobility* by using mobile objects. While the first relies on the ability of the network to locate a mobile terminal, the second addresses the ability to maintain the personal environment and transparently access network services. A particular challenge is to separate the application implementation from the underlying host environment. In *NetChaser*, an agent framework, *Autonomous Remote Cooperating Agents (ARCA)* [19], is used to run various agents that assist users in accessing information services and attempt to predict future user actions. While NetChaser deploys agents to integrate the personal environment into existing distributed systems and installed applications, there are mobile code approaches that deliver software on-demand to mobile terminals. As an example, *Achilles* provides a software delivery architecture that checks if available resources of a mobile computer are sufficient to run a selected application and when it is suitable to update installed software components [202]. Its main goal is the transparent provision of a customized user environment that supports disconnected operations of mobile clients. A step further towards personal mobility and pervasive personal computing is proposed with the *Internet Suspend/Resume (ISR)* approach [315, 316]. It separates a virtualized computing environment from the physical computing system by using *Virtual Machines (VM)*, such as *VMWare* or *Xen* [26]. Similar to the suspend/resume approach of modern operating systems; the user may suspend a VM anytime if he or she wants to move to another computing system. The captured execution state is then transferred to a remote storage system and can be retrieved by any target system that has been prepared to do so, i.e. by installing VMWare and the ISR software. The user can resume the VM and continue with his or her work at the point the execution has been suspended. Besides always transferring the entire VM state, there are variants that dynamically synchronize local changes of the VM with the remote storage to minimize the transmitted data, e.g. by intercepting disk operations. Another variant is the use of a mobile storage device to transport the suspended VM, such as implemented by *IoMega v.Clone* [174].

In summary, community computing solutions focus on managed infrastructures, dedicated applications and well-known configuration scenarios. In contrast to enterprise computing, the administration is not performed by a single authority but by many. This leads to the invention of standardized specifications to support and simplify the interaction among different parties, e.g. by using SOAP and ACL. This simplification creates a harmonized application environment on top of the heterogeneous network infrastructure and enables application developers to create uniform applications and services for a variety of computing systems, e.g. by using Rich Client Platforms. Concerning the support of cross-platform operation, community computing solutions cannot rely on a central management like in enterprise computing. Therefore, the computing nodes are separately managed using previously negotiated parameters, e.g. contacting the same UDDI registry and downloading additional components from well-known remote code repositories. This allows distinct administrators to link their computing systems into the community system anytime without the need to coordinate management tasks with other administrators.

3.4.4 Public Computing

The final operating scenario is *public computing* with a cross-platform operating environment dynamically spread across distinct networked computing nodes in the *Internet*. There is typically no managed coupling of computing devices at all, and resources are alternatively employed when they become available [255], e.g. running personal applications on unmanaged desktop computers in an Internet cafe or supporting global distributed computing like in the SETI project [11]. The devices are set up by different administrators in separate ways, and the major difference to the prior solutions is the lack of a pre-determined goal for which the devices are used. Therefore, no task-specific applications or services are installed in advance but deployed to a target computing device and run as needed, e.g. using transferable code deployment units such as JAR files. Well-known solutions are illustrated below.

Global Computing Network. To process certain computational tasks like the analysis of drugs in cancer research or the computation of prime numbers, conventional Grid computing approaches may not be powerful enough. A particular solution is the interconnection of Internet computing nodes to form a "bigger" Grid; a *global computing network* [204]. While in a typical Grid network the number of computing nodes moves around few thousands of computing nodes, an Internet computing network may easily exceed several millions and include various types of computing resources. In this context, various approaches have been pursued towards wide-area computing, as in *WebOS*, *Legion* and *Globe* [363, 151, 152, 332]. The common idea was to map heterogeneous computing resources into a worldwide virtual computer and to develop a distributed computing system in which remote objects and computing resources can be used like local ones [365]. In recent years, service-oriented approaches with autonomously operating nodes have gained much more attraction for distributed computing projects. Instead of providing remote resource access on the network-level, the interaction is performed on the application-level by using proprietary or standard service protocols. Common design considerations of related public computing systems and distributed task processing are discussed in [305]. Nowadays, the underlying approach has become well-known as *volunteer computing*. Popular examples are *SETI@home, World Community Grid* and *GIMPS* that utilize the combined computing power for solving specific problems [11, 172, 146]. To this end, related client software packages are installed on distinct Internet computing nodes that connect to a central project server and retrieve a work package to solve. After processing, the results are sent back to the project server where they are assembled and evaluated along with further result packages received from other clients. In contrast to Grid computing, the Internet nodes of a *global computing platform* are not supervised and assigned by a central monitor and scheduler but are loosely coupled on a volunteer basis. Each client is free to join and leave the public computing network and to offer or revoke resources as it wishes like in *XtremWeb* [109]. A particular volunteer computing solution is *Berkeley Open Infrastructure for Network Computing (BOINC)* that differs from the first distributed network computing approaches by separating the infrastructure management and the computational processing [10]. It allows users to reuse the same BOINC client installation for supporting various distributed computing projects and eases the concurrent computation by sharing account and configuration settings. As a result, BOINC creates a cross-platform operating environment for multi applications and distributed computing purposes.

Peer-to-Peer Networks. Another popular approach for public computing is the establishment of a loosely coupled networking infrastructure following the peer-to-peer model and without imposing the use of a particular application system like BOINC or an Internet browser. In contrast, P2P solutions mainly focus on network communication and how separately managed and so far unknown peers can interact. There is no common application model except that every peer typically operates in a self-configuring and self-integrating manner, e.g. by announcing own peer services and discovering peer nodes without user intervention. *Sun JXTA* is another P2P solution that defines a set of open protocols and allows any connected device on the network to link into the JXTA virtual network and seamlessly communicate with other peers, hiding the programming language, operating system and underlying network protocol [140]. This separation of business logic and network communication greatly eases the development of P2P application in a heterogeneous and cross-platform operating environment like the Internet. In addition, JXTA comes with autonomic features, such as the self-configuration of the network communication due to firewalls and *Network Address Translation (NAT)*. From a general point of view, JXTA has become a set of standard protocol definitions for dealing with various P2P infrastructure management tasks in Java, such as peer discovery, peer binding and peer routing. Application developers are free to choose a suitable reference implementation or implement the JXTA protocol suite on their own, e.g. for the use along with a not yet supported programming language. While JXTA may be used to add P2P functionality for certain application scenarios, such as file sharing, it introduces a pipe-based communication model that complicates object-oriented communication used among distributed applications. As a result, JXTA represents a good choice for developing new P2P applications without starting from scratch but lacks support for interconnecting regular applications, e.g. transparently accessing distributed objects. Nevertheless, P2P infrastructures may be also used to organize the peers and its resources in a wide-area computing environments. For example, *Tapestry* and *Pastry* provide an overlay location and routing infrastructure that enables the location-independent interaction with distributed P2P resources [306, 394]. In *p2pCM*, a distributed component deployment model is introduced on top of a structured peer-to-peer overlay network [285]. It aims at enabling *Computer-Supported Collaborative Work (CSCW)* in a wide-area environment by deploying required software components via a P2P network.

Rich Internet Applications. While the invention of a multi-purpose application environment like BOINC supports distributed component deployment, spontaneous application execution and transparent device employment, it still requires manual installation of software packages before it can be actually used. For desktop computing scenarios, another approach is the use of commodity applications like regular Internet browsers that have been already installed and configured on the target computing system. A related solution is *Asynchronous JavaScript and XML (AJAX)* that relies on the built-in feature of Internet browsers to execute applications written in JavaScript and encapsulated in regular web pages [293]. The particular improvement over standard web pages is the continuous communication with the application server in the background and reacting to user input without submitting a distinct web form every time. The result is a responsive web interface that behaves almost like a regular desktop system, such as the cloud-based web desktop *eyeOS* [106]. A further extension of this idea is pursued by the *Eclipse Rich AJAX Platform (RAP)* project [97]. Besides using AJAX for client-side user interface processing, it aims to introduce a corresponding application model on the server side that is based on an OSGI application system, namely *Eclipse Equinox* [95]. This results in the combination of client-side and server-side technologies to create a seamless cross-platform operating environment by using the best concepts of both worlds. Following the *Model View Controller (MVC)* concept, RAP maps client-side AJAX user interface elements on server-side OSGI components and enables Java developers to focus on the Java business logic rather than on how to implement the required web logic for dealing with different browser features. Besides using built-in browser features, there are *Rich Internet Application (RIA)* solutions that rely on extra browser plugins like *Adobe Flash*, *MS Silverlight* and *Sun Java FX* [216, 238, 341]. They are typically installed on-demand when a related web page is visited the first time and mainly improve the GUI capabilities compared to a native web page, e.g. by adding new animation elements. While the browser-based computing approach has revolutionized the web and supports public computing by implementing a cross-platform operating environment based on commodity technologies, it is still a web computing approach. It cannot be used offline, and the client-side operation is limited to the browser environment and currently running application; no concurrent use of multiple applications on the same web page is possible. In addition, the programming model is different from regular applications in that it requires some effort to transform regular desktop applications [388]. However, concerning on-demand computing scenarios, RIA supports the shift from resource-centric to task-centric computing by providing the right application the user needs and, in case of AJAX, without installing any software package in advance.

As a result, public computing solutions have recently become quite popular due to the de-facto permanent connection of computing systems to the Internet and the increasing availability of unused computing resources at the same time. A particular characteristic is the unmanaged infrastructure in terms of system setup and application deployment which results in public computing solutions separating the concrete application from the specifics of the underlying platform, e.g. introducing a common-purpose computing environment like in *BOINC* and *XtremWeb*. Another option is the shift from a box-centric to a network-centric view and the specification of the service communication instead of application hosting, e.g. by defining an independent set of communication protocols like in *Sun JXTA*. Certainly, there are further public computing solutions that mainly differ in terms of programming language and programming model [293], such as *Adobe AIR*, *Google Web Toolkit (GWT)*, *Microsoft Silverlight* and *Sun JavaFX* [2, 144, 238, 341]. Most of them have been developed with the focus on separately deploying and running Internet applications. They typically lack support for concurrent hosting of applications across heterogeneous computing systems and self-managing integration of software components offered by different Internet sites.

3.4.5 Results

While single computing, enterprise computing and community computing are addressed by many approaches, such as *Eclipse Equinox*, *IBM On Demand Operating Environment* and *Globus*, the public computing scenario is still subject of current research and ongoing development. Related work like *Berkeley Open Infrastructure for Network Computing (BOINC)* aim to create a distributed computing environment on top of unmanaged and loosely coupled computing systems, e.g. by temporarily running desktop computers at home. The major advantage is the establishment of a global computing infrastructure in which everybody can plugin and remove his or her computer at any time, without actually dealing with the specific application deployment and operation. While this solution focuses on *remote task processing*, other approaches, such as the *Eclipse Rich Ajax Platform (RAP)*, address *local task processing* in the public space. The idea is to set up and run specific applications on managed server computing systems and to use web browsers on unmanaged desktop computers to display the application interface and access local data. The user gets the illusion of a pervasive application environment in which applications are run on the currently employed desktop computer. A further approach is to support *distributed task processing* by specifying the interaction model instead of the application model, like in Sun JXTA. Each peer provides access to its resources and services via standard communication protocols, but is free how to realize the functions, e.g. by using a virtual runtime environment like Sun JRE or executing native program code. On the one hand, this results in greater flexibility to integrate different peers and services in a heterogeneous environment on the fly. On the other hand, major issues of spontaneous cross-platform operation are not addressed and supported, e.g. distributed code deployment and dynamic software composition. They are partially addressed by mobile code systems, such as *AgentScape*, which support ad-hoc execution migration but lack features concerning environment customization.

3.5 Summary

In this chapter, the motivation for building an autonomic cross-platform operating environment suitable to support on demand Internet computing has been presented. Various cross-platform issues have been discussed by considering the challenges of application management in a heterogeneous network environment. The outcome was the presentation of an approach that addresses the desired features of a cross-platform operating environment and considers the challenges imposed by realizing it in an Internet environment. Next, the need for autonomic operation has been motivated due to the uncertainty of application requests, spontaneous use of heterogeneous resources and dynamic application of user profiles that cannot be addressed by manual management or one-for-all configurations. After having deduced essential self-managing features for cross-platform computing, a self-managing infrastructure has been presented that is based on the integration, automation and virtualization of computing resources. Finally, supporting solutions for cross-platform computing have been outlined with respect to the addressed operating environment, namely single computing, enterprise computing, community computing and public computing.

The principal outcome of this chapter is the feasibility of combining functional cross-platform features with self-managing behavior to address the features of ODIC and the challenges of an open and unmanaged network environment as found in a public computing scenario. The idea of separating the platform setup, the application configuration and the environment customization along with the introduction of a self-managing middleware decouples the tasks of the user roles of ODIC as elaborated in Section 2.3.3. From this point of view, the proposed *Autonomic Cross-Platform Operating Environment (ACOE)* is supposed to particularly support the spontaneous deployment and launching of Internet applications, as described in the vision in Section 2.4.3.

The review of supporting solutions, primarily with respect to their support of ODIC and implementing an ACOE has revealed that there are many efforts to address local, remote and distributed task processing in single, enterprise and community computing scenarios. For public computing scenarios as typically found in the Internet, however, only few solutions exist that are typically limited in supporting the elaborated cross-platform features and self-managing behavior. In addition, they are typically supposed to address a single kind of task processing only, e.g. BOINC enables remote task processing but is not well-suited for local task processing in a public computing scenario. As a conclusion, a suitable realization supporting all facets of ODIC, as presented in Section 2.4.2, as well as implementing the features of ACOE, as proposed in Section 3.1.3, is missing.

4. XDK - The Crossware Development Kit

4.1 Introduction

In this chapter, the realization of the *Autonomic Cross-Platform Operating Environment (ACOE)* proposed in Chapter 3 is presented. First, the system architecture and the Java realization of the *Crossware Development Kit (XDK)* are sketched and its deployment in a cross-platform operating environment is outlined. Then, the implementation of the core features, namely deployment, composition, hosting, customization, interconnection and migration, is described in detail. Parts of this work have been already published in journals and conference proceedings [267, 268, 269, 270, 271, 272, 273, 274, 277, 278, 279, 280, 281, 282], and carried on in [89, 162, 297, 318, 362]. Last, the implementation is reviewed and the results with respect to the project goal of ACOE are discussed.

4.1.1 System Architecture

The system architecture of the XDK follows the proposals for building a cross-platform operating environment (Figure 3.4) and the self-managing infrastructure (Figure 3.7) presented in Section 3.2.2 and Section 3.3.3. As illustrated in Figure 4.1, the layered system architecture is separated into the platform, the middleware and the application layer.

The platform layer represents the host-specific installation found on a target computing node. A suitable operating system has been installed by the administrator, e.g. MS Windows, and additional features like hardware drivers and database installations have been added to the installation. On top, a virtual runtime environment is set up that hides the actual node installation and provides a uniform interface to the platform resources. The middleware layer contains the core of the self-managing infrastructure components. An integration middleware complements the virtual runtime environment with additional features concerning cross-platform operation, e.g. code deployment units, application composition descriptions and virtual object interconnection. The layer feature automation contains the basic components for creating a self-managing infrastructure across distinct computing nodes. In fact, it represents the autonomic manager that controls the host resources provided by the integration middleware and establishes a virtual operating environment on top; hence it is an autonomic element altogether. As shown in Figure 4.1, the resulting control loop involves the following layers: integration middleware (managed element), the feature automation (autonomic manager) and the virtual operating environment (touch point). The application layer is located on top of the virtual operating environment and benefits from the cross-platform automation provided by the self-managing infrastructure.

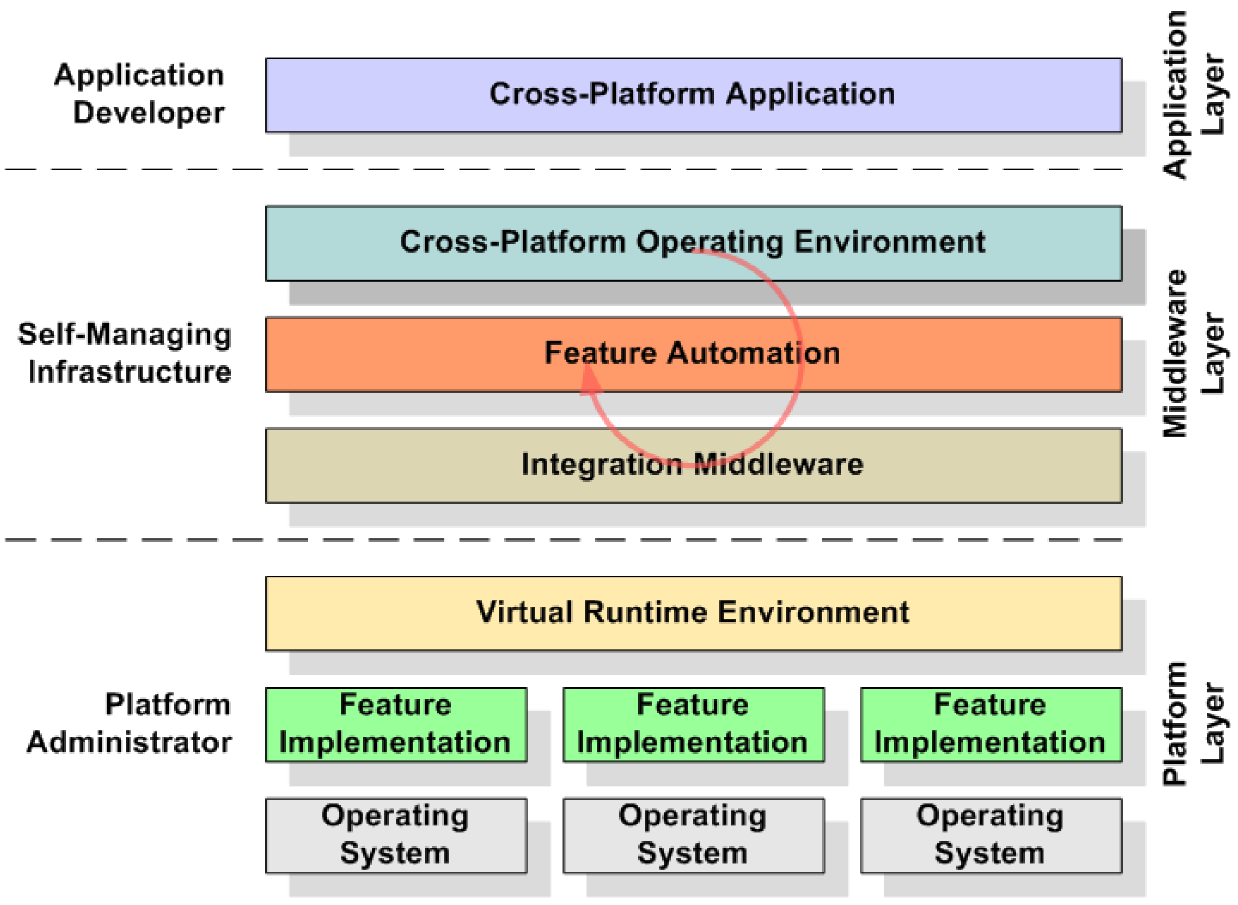


Figure 4.1: XDK System Architecture

An application does no longer have to be prepared to run on a specific target host but can be configured with respect to the uniform and self-managing features of the virtual operating environment. As a result, the system architecture combines the features of a cross-platform operating environment presented in Section 3.2.2 with the automation capabilities of the self-managing infrastructure described in Section 3.3.3.

4.1.2 Java Realization

The system architecture shown in Figure 4.1 is based on a virtual runtime environment that allows running the same program code on different host systems. As discussed in Section 3.4, there are actually two competing solutions suitable at present, namely Sun Java and Microsoft .NET, which both provide a fully-featured and matured virtual runtime environment. However, recalling the objective to establish a cross-platform operating environment consisting of hosts with heterogeneous operating systems, only Sun's Java is currently a working approach. Microsoft .NET is originally limited to Microsoft Windows platforms, although there are various open source projects on the way, e.g. Mono, which allow running .NET application on different operating systems, the feature and installation support are still limited [257]. Moreover, Java is actually the preferred runtime environment for running cross-platform applications in the Internet, e.g. Java applets and servlets. Consequently, the Java Standard Runtime Environment (J2RE) is chosen to realize the ACOE, as shown in Figure 4.2.

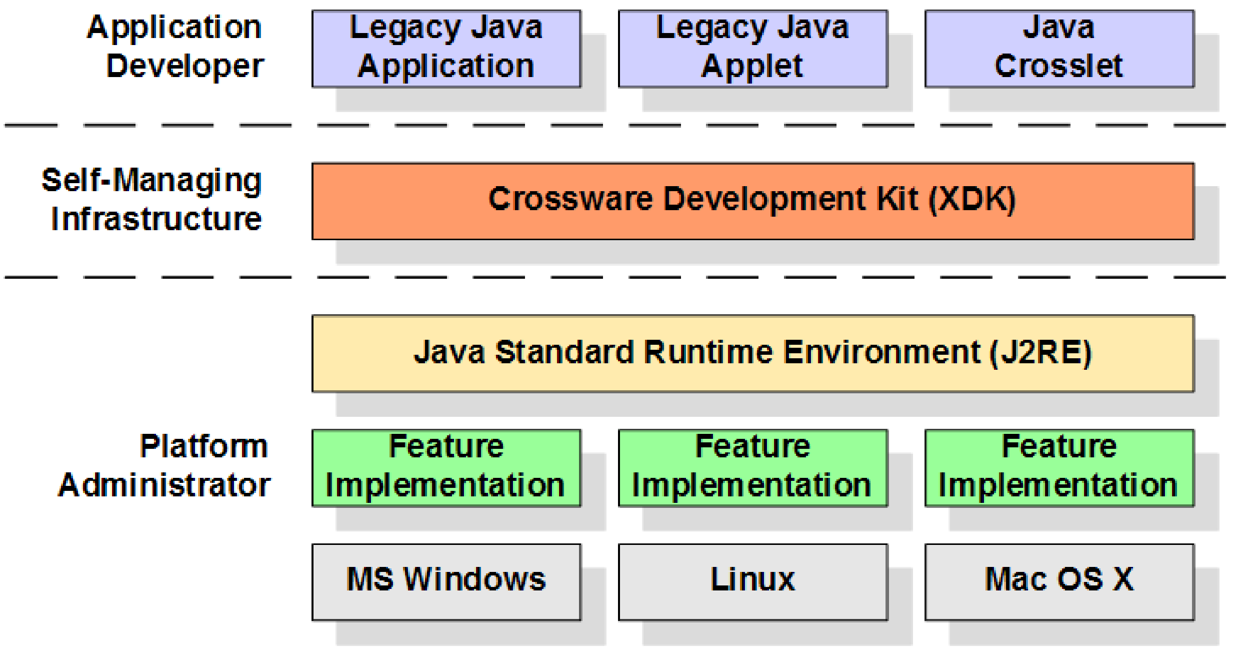


Figure 4.2: Java Realization of the Autonomic Cross-Platform Operating Environment (ACOE)

The middleware layer from Figure 4.2 is implemented by the Crossware Development Kit (XDK) whose implementation is described in the following sections in detail. In short, the XDK represents the autonomic element that manages the platform resources and provides self-managing features to the cross-platform applications. The latter can be a legacy Java application, a legacy Java applet or a Java crosslet that represents a new type of cross-platform application introduced by the XDK and described in Section 4.2.1. The basic idea is to support existing applications as well as to offer and implement advanced features arising from the interaction of the crosslet and the XDK.

4.1.3 Components

Before the realization of the XDK is described in detail, a brief overview of the components targeting the ACOE is given. Recalling the feature list in Section 3.1.3, a corresponding implementation has to consider various issues, as illustrated in Figure 4.3.

The XDK addresses distributed code deployment by introducing *Self-Descriptive Crosslets* and *Java Class Collections*, described in Section 4.2. The dynamic software composition is handled by *Java Class Spaces* and *Java Loadable Modules*, described in Section 4.3. In Section 4.4, *Adaptive Resource Broker* and *Java Task Spaces* are proposed for shared application hosting. This is followed by the illustration of the *Application Execution Engine* and *Roaming User Profiles* in Section 4.5 used for the pervasive environment customization. Next, *Java Method Streams* and *Java Object Spaces* address virtual object interconnection, as illustrated in Section 4.6. Finally, the support of ad hoc execution migration is described in Section 4.7 by presenting *Java Thread Controller* and *Java Execution Units*.

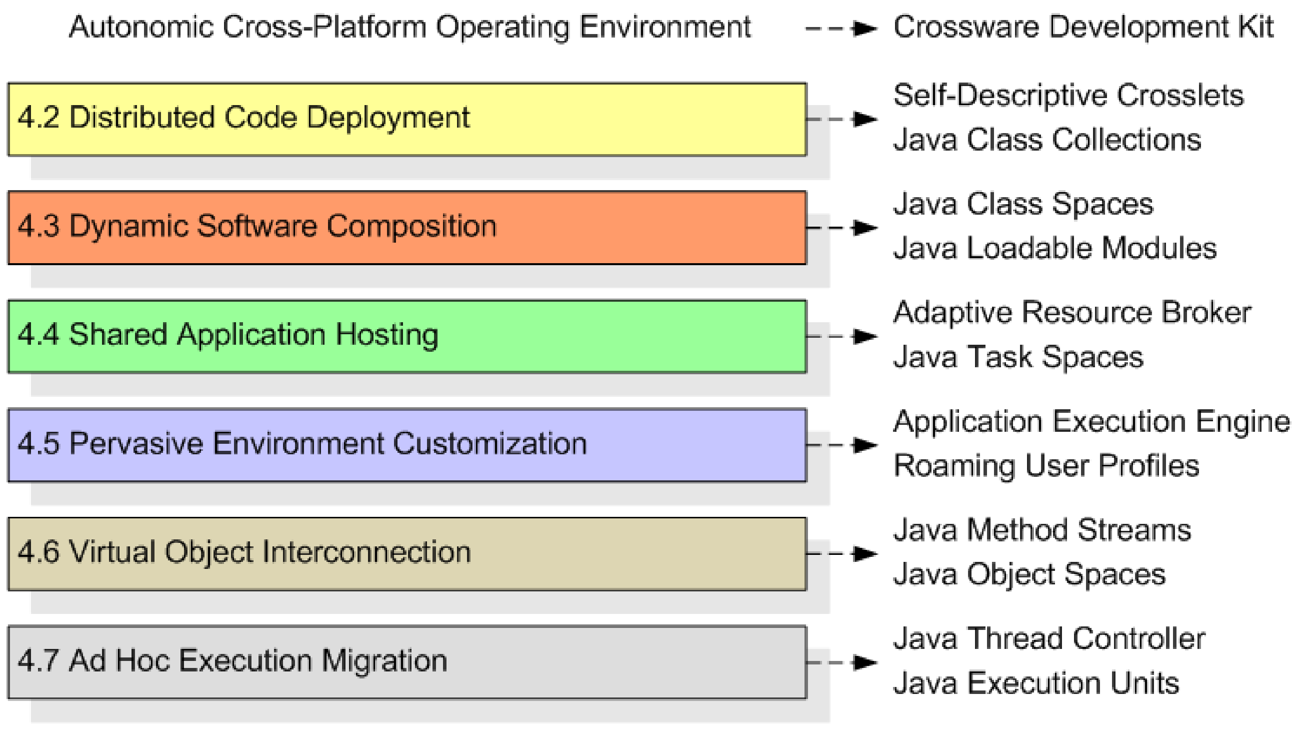

Autonomic Cross-Platform Operating Environment
Crossware Development Kit
4.2 Distributed Code Deployment
Self-Descriptive Crosslets
Java Class Collections
4.3 Dynamic Software Composition
Java Class Spaces
Java Loadable Modules
4.4 Shared Application Hosting
Adaptive Resource Broker
Java Task Spaces
4.5 Pervasive Environment Customization
Application Execution Engine
Roaming User Profiles
4.6 Virtual Object Interconnection
Java Method Streams
Java Object Spaces
4.7 Ad Hoc Execution Migration
Java Thread Controller
Java Execution Units

Figure 4.3: Feature Mapping

4.2 Distributed Code Deployment

The employment of computing resources requires the separate installation of software on spatially administered computing nodes. This is usually performed by the roles of the assembly deployer who copies the software from a well-known code repository and the system administrator who runs the specific software installation program. In a distributed environment, various code repositories like public FTP mirrors are concurrently employed by component providers and assembly deployers to propagate component and application assemblies, respectively, e.g. plugins in packaged in single software libraries and application installers that subsequently download necessary bundles during installation. In a self-managing cross-platform operating environment, a particular problem is the automatic selection of suitable assemblies from various code repositories without user intervention. In Section 4.2.1, *self-descriptive crosslets* are introduced for packaging and describing code assemblies [280, 355]. The aim of the new deployment approach is to enable the concurrent shipping of assemblies to different code repositories and the dynamic selection and download of appropriate bundles according to the application and platform configuration, e.g. locating a matching database driver to access the locally installed database. For Java applications, an additional problem is the native class loading approach that does not operate in terms of Java assemblies, e.g. Java archives (JAR files) but classes found in the CLASSPATH setting. In Section 4.2.2, a refinement of regular Java class loading is presented using *Java class collections* [270]. It is shown how they support the definition of code assemblies in Java by decorating regular Java archives and how Java class loading is modified to transparently select and load required code assemblies in a self-managing way.

4.2.1 Self-Descriptive Crosslets

The term *crosslet* is introduced in this thesis to denote a novel self-descriptive code deployment unit that allows organizing the code shipping in a self-managing way. Though the approach is implemented in Java and primarily intended to be used with Java applications, it is shown how crosslets support the self-managing selection and deployment of native application assemblies, e.g. retrieving Linux executables matching the configuration of the target computing system.

4.2.1.1 Motivation

A software application is typically composed of smaller parts, e.g. libraries, modules and components that are provided by component developers in separate code packages [see Section 2.3.2]. In a traditional deployment scenario, the application assembler selects appropriate software components, adds an application configuration and creates an application assembly suitable for a prearranged computing system, e.g. a MS Windows executable bundled with all required DLLs and often supplemented with a specific application installer. In turn, a system administrator selects an application assembly appropriate for the target computing system, e.g. by ordering a related installation CD, and performs the required installation steps, e.g. preparing the computing system, running the application installer and choosing the installation options suitable for the present computing system.

While this approach works fine in a managed environment and with well-known application installations, it has drawbacks in a cross-platform operating environment where application installations not known in advance co-exist on heterogeneous computing systems, e.g. installed by different users of the same computing system. First of all, applications are deployed and installed in an all-or-nothing fashion. There is usually no way to reuse software components from parallel application installations that causes the deployment the same component another time. This wastes disk space and increases the transmission overhead in case of remote installations, e.g. via the Internet. Another drawback is the administrator-driven installation scheme that assumes the manual selection of software components with respect to the target computing environment. The replacing and mixing of software components from different sources is not generally supported and only possible for experts. A further issue is the lookup of suitable application software to install. There is not only one code repository but several may be offered from where missing software components can be retrieved. A user has to manually query each one for new and updated software releases which is not only a tedious task but may be practically impossible in a large scale environment.

4.2.1.2 Features

The goal of the *Self-Descriptive Crosslet* approach is the introduction of a cross-platform deployment system for distributing and resolving software packages in a self-managing way. The major features are as follows.

Self-Descriptive Deployment Unit. An essential feature is the introduction of comparable deployment units that can be used by a self-managing deployment approach to organize and evaluate deployment units in a uniform way and regardless of their content, e.g. determining deployment dependencies and identifying compatible variants in the local deployment cache before actually downloading the same deployment unit another time.

Distributed Code Repositories. In a cross-platform operating environment, there are various authorities that may concurrently introduce different releases of the same component and upload them to multiple code repositories. By using crosslets, a self-managing deployment approach can access distinct code repositories and query them for missing components, e.g. resolving the dependencies of a component found in another repository.

Self-Managing Code Deployment. A particular concern is the customizable selection and resolution of required components according to the requirements and capabilities of the requesting application and the current hosting environment, respectively. The self-managing code deployment approach evaluates the current deployment scenario and adapts the deployment query, e.g. by dynamically selecting an updated release of the originally requested component.

Transparent Use. The deployment process can be dynamically invoked and altered without affecting the current application execution, e.g. switching from one code repository to another on-the-fly or selecting a compatible variant already installed on the host computing system. The self-managing deployment procedure is hidden from the application and the user as well, e.g. how to resolve the dependencies of an application plugin.

Legacy Application Support. The self-managing deployment approach is able to deploy legacy applications without modification, e.g. packaging Java applications without recompilation but by adding particular deployment descriptions. In addition, native executables, such as MS Windows programs, can be packaged as well and decorated with configuration statements for deployment on suitable MS Windows computing systems only.

4.2.1.3 Approach

The idea behind the approach is to adjust the dynamic composition of applications in a self-managing way according to the current deployment scenario. Application requirements, platform configurations and available code assemblies are evaluated and the best matching assemblies are selected and used for application composition. To this end, the approach introduces a self-descriptive assembly unit called *Crosslet* that acts as a physical container for custom resources, e.g. code archives or image files, and may be individually deployed to distributed crosslet repositories [277]. Every crosslet contains a crosslet description to control the deployment of the crosslet by specifying its properties and dependencies. If an application is to be deployed and run on a computing system, the required crosslets are dynamically selected and retrieved from crosslet repositories, as illustrated in Figure 4.4.

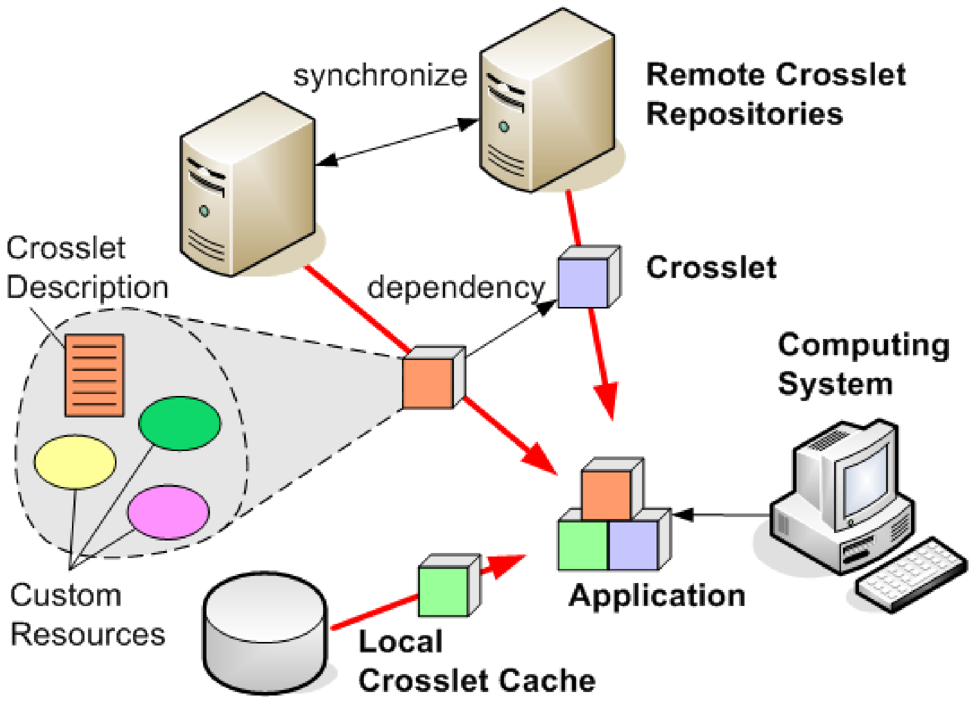


Figure 4.4: Application Deployment using Self-Descriptive Crosslets

The crosslet repositories are separately run by different authorities and may be connected to synchronize their crosslet stores, e.g. for forming a master and backup repository installation. While a computing system can retrieve all crosslet descriptions from every repository and resolve the required crosslets on its own, the preferred way is to submit a list of required crosslets, e.g. by specifying the crosslet properties, to a single crosslet repository. This repository will then resolve the crosslet dependencies by querying the other repositories. Thus, there is no need to update every computing system with the information of all installed crosslet repositories and system administrators are enabled to set up a single connection point to a varying crosslet deployment infrastructure.

4.2.1.4 Realization

The actual application deployment on a computing system is performed by a *self-managing crosslet installer*. It evaluates the application description and the platform configuration to select and retrieve the application crosslets best suitable for the current computing system, as shown in Figure 4.5.

The deployment procedure of application crosslets is as follows. First, the installer reads the crosslet dependencies from the application description (1) and checks the crosslet cache for matching crosslets (2). Next, the platform configuration provided by the system administrator is read (3) and a crosslet query is defined (4) while considering both the application requirements and platform configuration as well. The crosslet repositories are queried and the crosslet descriptions of matching crosslets are returned in a list (5). Next, the dependency resolution starts and tries to determine which crosslets should be actually retrieved. Finally, the list of resolved crosslets is returned and the crosslets are downloaded from the crosslet repositories (6). To adjust the self-managing deployment process, a deployment policy can be defined to modify the crosslet selection and installation according to additional parameters and constraints, e.g. excluding crosslet repositories that failed in the past or prefering crosslets from a certain provider.

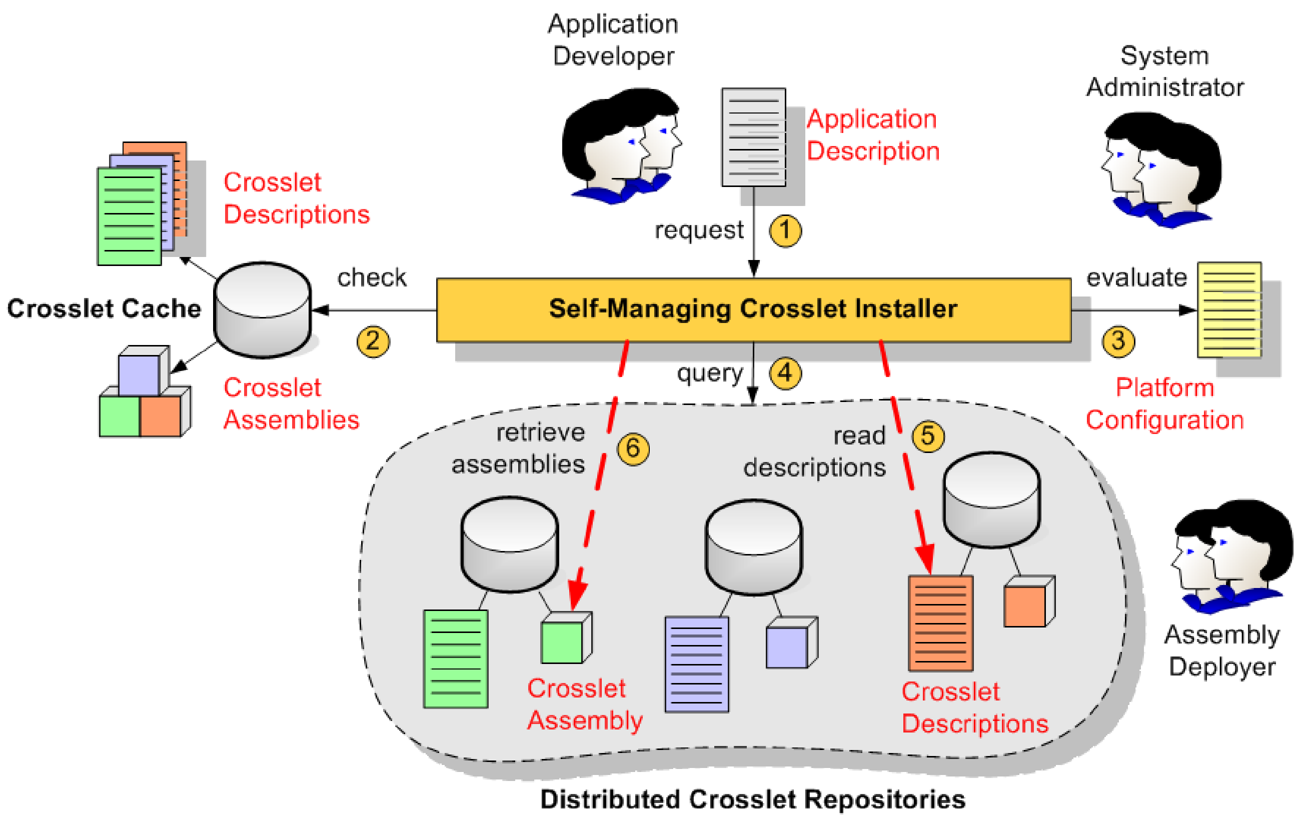


Figure 4.5: Self-Managing Crosslet Installer

A particular objective for the realization of the crosslet approach is the transparent application in regular development and deployment scenarios. For this reason, existing file formats and common tools to package a crosslet are used. A ZIP-based file format is chosen where code bundles are encapsulated and described using various configuration files. Similar to a WAR file (Web Archive), a XAR file (Crossware Archive) introduces a specific file organization, as shown in the example in Figure 4.6.

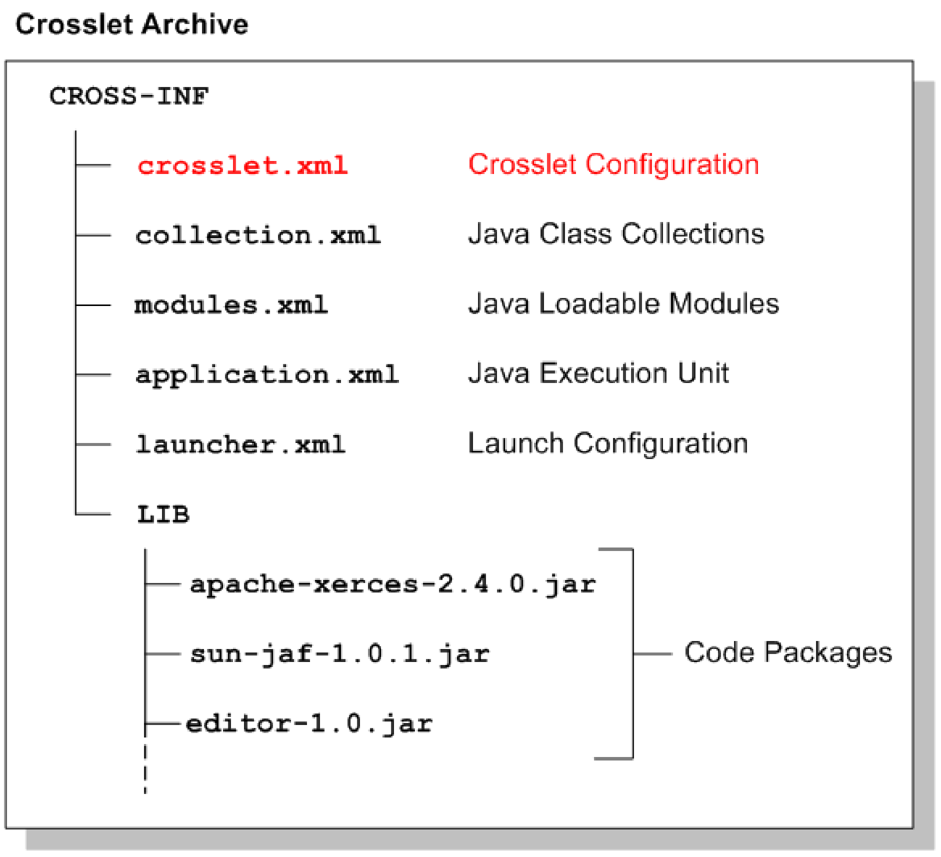


Figure 4.6: File Organization within a Crosslet Archive (XAR)

The XAR file contains a particular folder CROSS-INF in which specific configuration files are located and a sub directory LIB that can host various code packages. The mandatory XML configuration file `crosslet.xml` describes the crosslet content and is mainly used to identify the crosslet and its deployment dependencies, e.g. which crosslets have to be retrieved before this crosslet can be enabled. Additional configuration files, such as `collection.xml`, `modules.xml`, `application.xml` and `launch.xml` can be optionally added and are used to refine the composition procedures after the deployment phase [see Section 4.2.2, Section 4.3.2, and Section 4.5.1]. An excerpt of a crosslet configuration is shown in Figure 4.7.

The XML crosslet configuration file `crosslet.xml` starts with a root node `crosslet` and a related attribute `id` that marks this crosslet with a unique identifier, e.g. `archiver-1.3.0-070103.2340`. The format of the identifier is free as long as the id string is unique. The remaining configuration is divided into three sections, namely `properties`, `dependencies` and `command`, as shown in Figure 4.7. The sections `properties` contains key-value pairs of various meta-data properties, e.g. the identifying name of the crosslet, the release statement for comparing other variants of the same crosslet, an informal description and a directory statement for grouping related crosslets when the crosslet properties are displayed in a deployment browser.

```
<crosslet id="archiver-1.3.0-070103.2340" >

    <properties>
        <property name="name" value="crossware-archiver" />
        <property name="release" value="1.3.0" />
        <property name="description" value="Archiver" />
        <property name="directory" value="/crossware/odix" />
    </properties>

    <dependencies>
        <crosslet>
           <property name="name" value="crossware-odix" />
           <property name="release" value="2.1.0-dev1+" />
        </crosslet>
        <platform>
           <property name="os.arch" value="x86" />
           <property name="os.type" value="windows" />
        </platform>
    </dependencies>

    <command type="Install" title="Install Application">
        <launcher id="{DF18CB03-D0B3-46cb-8499-A8264D9738FB}" />
    </command>

</crosslet>
```

Figure 4.7: Example of Crosslet Configuration File

Apart from the mandatory property `name`, all properties are optional and can be extended with any other key-value pair to refine the crosslet meta-data. In turn, the `dependencies` section uses these meta-data properties to refer required crosslets by defining corresponding properties to match against. In the example, when the crosslet `archiver-1.3.0-070103.2340` is deployed a crosslet with the name `crossware-odix` and the release `2.1.0-dev1` or newer has to be retrieved as well. If there is more than one crosslet containing the matching property values, the most recent crosslet will be used. By specifying additional properties, the component deployer can refine the deployment dependencies and thus the deployment process. Another option is to use the crosslet id that uniquely refer a certain crosslet but will not allow the system to select a compatible variant if possible. In addition to refer other crosslets, the deployer could also constrain the deployment of the crosslet to platforms with certain features. As an example, by using `os.type` and the value `windows`, one can bind the deployment of the crosslet to MS Windows platforms only. It should be pointed out, that the configuration file may be extended with further property statements that refine the deployment process, such as screen resolution or the amount of installed main memory. Furthermore, the current property matching procedure is rather simple and does only compare related properties by regular string matching or proprietary char sequences like the plus sign in the release statement meaning this or a more recent release. However, the evaluation of the crosslet configuration file can be adjusted by plugins that could rely on regular expressions or introduce a complete different crosslet matching scheme. Finally, the `command` section can be used to add references to particular launch configurations found in the launch configuration file `launch.xml`. In the example, a launch configuration is referenced that will be evaluated when this crosslet is to be installed. Further crosslet commands can be added, e.g. for updating and uninstalling an application.

The current realization of the autonomic code deployment logic is part of the XDK and implemented in Java. Nevertheless, the crosslet deployment units and the XDK can also be used for the autonomic deployment and launching of native applications. To this end, the executables and required extra files are packaged into a ZIP file that will be in turn put into the crosslet. A particular application launcher evaluates the crosslet after deployment, unzips the executables into a temporary directory and starts the application a separate process like any other native executable. In case of Java programs, various applications may also be launched and organized within the same Java Virtual Machine, as will be shown in Section 4.3.1 and Section 4.4.2.

A simple crosslet repository can be set up by using a regular file system and storing the deployed crosslets as serialized files, e.g. often used for installing a local crosslet cache. Another option is to use a database installation and store the crosslet description in database tables for better handling. Furthermore, a crosslet repository may be installed as a web service that can be easily accessed via the Internet. When uploading a crosslet to a crosslet repository, the crosslet description is extracted and stored separately from the actual crosslet unit. This enables the querying of crosslets without reading possibly large crosslet files during operation later on. In addition, a computing system can request the crosslet descriptions and select crosslets before actually downloading them.

4.2.1.5 Autonomic Application

The present crosslet deployment approach can be used to implement *Autonomic Code Deployment* for spontaneous launching of applications by retrieving and installing required crosslets automatically, as shown in Figure 4.8.

If the application launcher receives an application launching request, it delegates the retrieval and installation of suitable crosslets to the crosslet manager. Various crosslet repository handlers, that have been configured by the platform administrator during system setup or discovered during runtime, may be accessed by the crosslet manager to query missing crosslets.

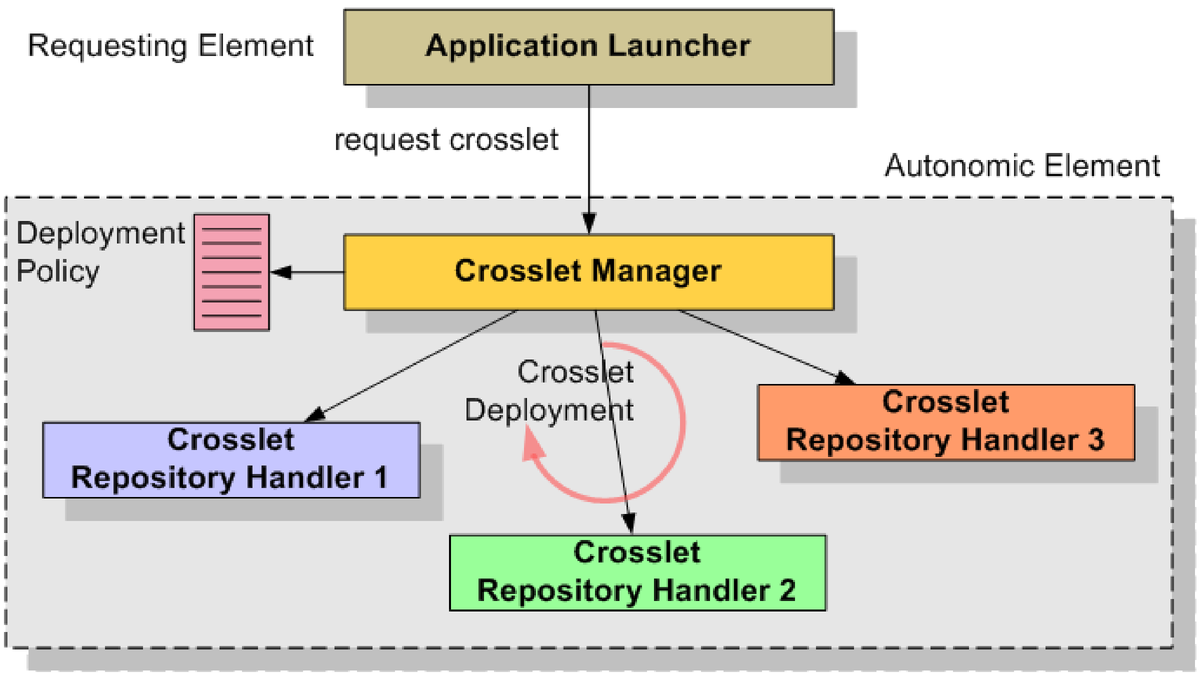


Figure 4.8: Autonomic Code Deployment

Code deployment is performed in an autonomic manner by evaluating a given deployment policy that specifies various rules for selecting and downloading required crosslets. From this point of view, the application launcher represents the requesting element whereas the crosslet manager acts as the autonomic manager that controls various crosslet repositories, the managed elements, to query and retrieve needed crosslets. For example, the crosslet manager could discover online crosslet repositories by querying well-known repositories and prefer crosslet repositories in the Intranet or reachable via secure connections *(self-configuration)*. Before actually launching an application an administrator could have specified to always check for updated crosslets located in repositories in the Intranet or ignore crosslet repositories that have failed in the past due to poor network connectivity *(self-optimization)*. Another scenario is the movement of a nomadic user to another computing system and the synchronization of application launcher profiles [see Section 4.5.1]. The crosslet manager is able to detect unresolved crosslet dependencies on the new computing system and try to retrieve the missing crosslets in the background *(self-healing)*. Finally, the crosslet manager checks all crosslet installation tasks and prevents the installation of crosslets that may compromise the installation, e.g. replacing system crosslets or downloading crosslets lacking a trusted security certificate *(self-protection)*.

4.2.1.6 Related Work

There are many Java deployment approaches available that mainly differ in the way how Java classes are packaged, distributed and retrieved. Various aspects like scalability and complexity may be considered to select a suitable deployment solution [351]. In the following section, related work is reviewed with respect to the applicability in a cross-platform operating environment.

Native Java Deployment. The native Java deployment approaches are basically characterized by using the Java system class loader. A simple option is to put the Java classes in a directory structure following the package hierarchy and set the CLASSPATH accordingly. This approach is suitable for development time but fails to support remote code distribution. A refinement is the use of a Java archive (JAR) that is built by packaging the directory structure into a single file [367]. On the one hand, a JAR file can be easily distributed and used to add extra information about the contained classes, such as version statements in the manifest file. It is compatible with each legacy JVM and is therefore well-suited for a cross-platform operating environment. On the other hand, it lacks support for dynamically configuring the composition, customization and execution. Once started, the JVM cannot be easily reconfigured to consider additional JAR files that were not added to the CLASSPATH. The customization of an application is not possible without modifying the JAR files, and the runtime configuration is not addressed at all by this approach.

Java Component Frameworks. There are framework approaches that emerged from standardized application scenarios, such as *CORBA Components* [232] and *Java Servlets* [171]. They address specific deployment and composition scenarios that are defined by the framework implementation. For example, web modules are packaged in *Web Archives (WAR)* files that are JAR files containing particular configuration directories and files, such as Java classes, HTML and XML files. They are supposed to be exclusively used by a servlet engine for the deployment of Java servlets. The *Enterprise Java Beans (EJB)* approach introduces *Enterprise Archives (EAR)* that adds an additional abstraction level to group various WAR and JAR files into a single entity [242, 336]. This makes it easy to reuse components in new J2EE applications and distribute them to another application server. Both approaches focus on the support of specific server-side application scenarios and are not suitable for different kinds of applications, e.g. legacy Java desktop applications. While a WAR file does not basically differ from a JAR file concerning deployment configuration, an EAR file can be used to configure the composition, customization and execution of a J2EE application. Though it separates the concerns of application deployment and supports different user roles, such as application assembly, it is not able to dynamically modify the deployment and composition process during runtime. It lacks support for remote code repositories and always bundles the code along with the configuration files as a single entity. A popular deployment approach is introduced by the OSGI service platform [262] that focuses on the installation and management of self-descriptive software components. It introduces so called *bundles* as deployment units that are based on legacy JAR files with extended manifest files within. The OSGI approach relies on a well-known deployment scenario where bundles are provided at specific locations from which they are downloaded during the dependency resolution process if a new component is to be installed. In this scenario, distinct bundles cannot be dynamically selected and retrieved from concurrent repositories. Furthermore, the OSGI deployment approach is designed to particularly support the dynamic composition of OSGI applications and is not feasible for the deployment of legacy Java applications, e.g. due to the need of an OSGI runtime installation.

Dynamic Code Deployment. A dynamic approach is specified by the *Java Network Launch Protocol (JNLP)* and used by various implementations for client-side deployment, such as *Sun Java Web Start*, *Netx* and *Object Component Desktop* [383, 254, 260]. Instead of distributing code and application configuration as a single unit, a JNLP configuration file is retrieved from a remote application repository and used to dynamically configure the deployment process. The approach supports local caching of downloaded JAR files and checking for updated versions that are transparently downloaded when the application is started next time. In addition, it supports the configuration of the application composition by introducing particular server-side JNLP handlers and the parameterization of the application execution. The dynamic selection and configuration of a suitable runtime environment is possible. Although the JNLP approach supports many issues of dynamic application deployment in a cross-platform operating environment, it is basically limited to a fixed deployment scenario, e.g. using well-known JNLP repositories like the upcoming *Sun Java Store* [346]. The distribution and composition configuration is tightly coupled, and there is no way to dynamically include or query other code repositories. There is no support for self-managing customization of the deployment process such as the selection of the most appropriate component according to application requirements and platform capabilities.

Custom Networked Deployment. There are several custom deployment approaches that address different application scenarios. *Deploy Directory* [84] and *Power Update* [296] are designed to manage auto-updating of Java clients and use a proprietary deployment protocol. The approaches are able to customize the deployment process but are not supposed to be used in application scenarios different from local task processing. A similar solution is *Software Dock* [160]. It enables cooperative software deployment by introducing particular servers, release dock and field dock, which represent software producers and software consumers, respectively. Agents implement the actual software deployment functionality and use the servers to deploy and to retrieve software systems.

While this approach introduces an advanced software deployment infrastructure, it heavily relies on a well-known deployment scenario and specific agents to perform certain operations, such as checking for software updates. The approach supports the installation and removal of separate software systems but does not explicitly address custom application composition. In the context of nomadic and mobile applications, there are approaches that extend mobile agent solutions to take along the related Java software components from one host to another, e.g. *TACOMA* [335]. For pervasive application scenarios, the component deployment may be coupled with the movement of the user, e.g. by using RFIDs (radio frequency identifiers), and performed in a self-managing way, as in the *Hydra framework* [313]. Both, TACOMA and Hydra are designed to be mainly used in the addressed deployment scenarios and cannot easily be adopted by legacy applications. They introduce specific component assemblies or transfer components directly between two nodes without using an assembly entity at all. A further approach is *SATIN* [391] that provides a lightweight component model supposed to be used in mobile devices. It adapts itself to changing requirements and enables self organization based on logical mobility and the introduction of *Logical Mobile Units (LMU)*. An inherent drawback is the compulsory use of the related component model that actually turns this approach unfeasible for the deployment of legacy Java code. Another custom deployment approach is represented by *SmartFrog* [137]. It defines an application as a collection of possibly distributed components that are automatically deployed and configured. A particular specification language is introduced to define the lifecycles and dependencies of components and how they should be deployed, run and connected. The major drawback of this approach is the encapsulation of each instantiated component in a separate process or JVM. A certain middleware approach, such as RMI, must be used to connect the components. Moreover, there is no way to share commonly required software libraries which increases the resource requirements.

To summarize, native approaches are commonly used to encapsulate and distribute application classes, such as Java archives (JAR), but do not support distributed scenarios. Application framework approaches, such as Java servlets and OSGI, add particular features but are typically limited to well-known application scenarios and custom programming models. Dynamic approaches based on JNLP are able to customize the deployment process but depend on administered assembly repositories. Custom approaches tend to be specific for a certain application scenario, fail for different ones or cannot be easily adopted like the Hydra framework. Moreover, they typically require particular installations in advance and lack support for legacy Java applications, such as SATIN. From this point of view, the crosslet approach represents a self-managing deployment solution while it still supports legacy Java applications and class assemblies.

4.2.2 Java Class Collections

Java class collections support the definition of code packages based on regular Java archives and for use with legacy Java applications. The following sections show how existing Java code archives are decorated with various metadata defining the properties and dependencies of the contained classes. The realization using a particular Java class loader is described and the application of the approach for the self-managing selection of suitable code packages from remote code repositories is illustrated.

4.2.2.1 Motivation

The original design of the Java Runtime Environment (JRE) is based on dynamic Java class loading and a Java Virtual Machine (JVM) that instantiates Java classes as they are requested by locating, loading and resolving the related byte code [223]. This reduces the startup time. Later on, the memory allocation since only required classes are loaded and resolved. The native JVM comes with a so called *system class loader* that locates the byte code using an environment variable CLASSPATH and the *fully-qualified class name (FQCN)* of the required classes (composed of the package name and the class name itself) [94, 367]. The system class loader is able to load classes deployed into directories of the local file system or stored in Java Archives (JAR). In addition, custom class loaders can be installed that are free in the way the byte code is located and loaded, e.g. loading the JAR files from a remote repository via the Internet.

Typical custom class loader implementations usually change the way classes are *loaded* and sometimes also how they are *located*, but not how they are *selected*. They still use only the package and class name. Thus, it is not possible to configure from where which classes should actually be used when composing a specific application. In contrast, the classes are selected in the sequence as they are found which is not always correct, e.g. when an application is composed of classes from various Java Archives that contain several classes having the same class name. A related problem is the separated handling of distinct class loading requests which affects the resolution of class dependencies. Since the origin of a class is not remembered, associated classes may be loaded from a different location, e.g. from another JAR file than the referencing class, and an incompatible class variant may be possibly resolved and initialized.

4.2.2.2 Features

The *Java Class Collection* approach is aimed at introducing a Java component assembly that enables the self-managing deployment and composition of Java applications in a shared hosting environment. The major features are as follows.

Java Class Grouping. Java classes can be virtually grouped by selecting Java classes whose FQCN match a regular expression, e.g. all classes of the package `java.lang`. In contrast to the regular class loading approach, a particular class loader evaluates the JCC configuration and can ensure to load associated classes from the same origin. In addition, JCC allows deployers to create class groups spanning various class repositories, e.g. grouping classes contained in several JAR files.

Custom Assembly Tagging. The JCC can be decorated with custom assembly tags that may be used to select Java classes not only by the FQCN but also by extra properties, e.g. vendor and release tags. A related feature is the option to evaluate the properties without actually downloading the entire JCC from a remote code repository that eases the self-managing deployment in a cross-platform operating environment.

Adaptive Dependency Resolution. The declaration of dependencies and compatibilities among JCC by using an extensible description scheme is another feature. An application framework can evaluate the descriptions and determine the required Java classes before the application is actually launched, resolve the related JCC and, if the composition cannot be performed, try to resolve the dependencies in a self-managing way, e.g. by choosing compatible Java classes from another JCC.

Existing Class Repositories. A keystone of the JCC approach is the use of existing class repositories, like *Java Archives (JAR)* or *Open Services Gateway Initiative (OSGI) bundles*, which can be referred in a JCC configuration without modification or repackaging. The contained Java classes may be loaded from the local file system, a remote server or from any other origin by using a custom class loader plugin, e.g. from a database.

Legacy Java Runtime Environment. The JCC approach can be used with any legacy Java Runtime Environment and does not require any modification of the application implementation; hence the approach does not change the standard Java programming model that is especially important for developers and the reuse of existing code. The JCC configuration can also be used to create a corresponding JAR file for launching any regular Java application.

4.2.2.3 Approach

A new class loading approach is invented that is no longer relying on the location of a native Java classes or JAR files. Instead, so called *class collections* [268] are introduced that represent groups of classes that can be randomly defined and decorated with custom properties, as shown in Figure 4.9.

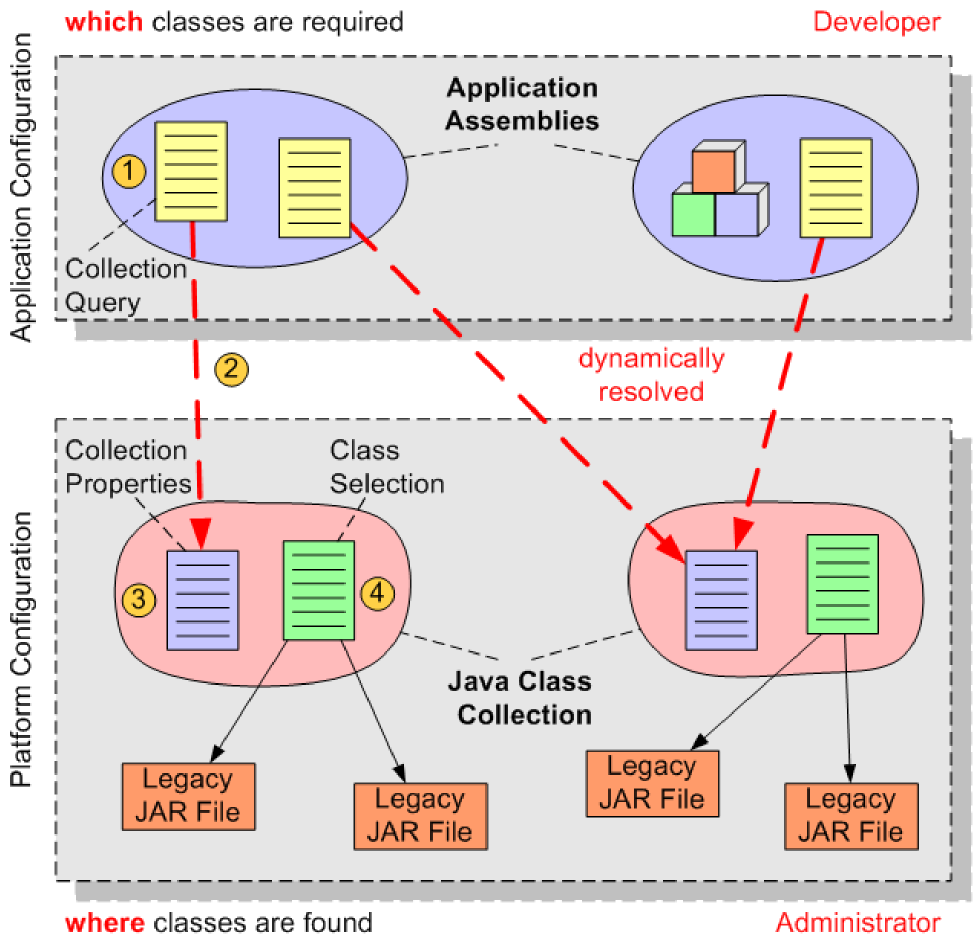


Figure 4.9: Class Grouping using Java Class Collections

The system administrator provides a list of available Java class collections in the platform configuration. Every class collection contains a configuration file defining the properties (3) of the collection and *where* the associated Java classes (4) can be found, e.g. in JAR files stored on the computing system or downloadable from a web server. The application developer specifies in the application configuration of an application assembly *which* classes are required to run the application. Particular collection queries are defined and used to resolve matching Java class collections on the current computing system. Both application and platform configuration can be individually created and are dynamically evaluated when an application is started to resolved the required Java classes. There is no need for application developers to know the platform configuration of the target computing system and vice versa, the platform administrator can define the platform configuration without yet knowing the applications to be run on the computing system.

4.2.2.4 Realization

To evaluate the Java class collections during runtime, a custom class loader is introduced that handles class loading requests by evaluating the collection properties of the configured Java class collections, as shown in Figure 4.10.

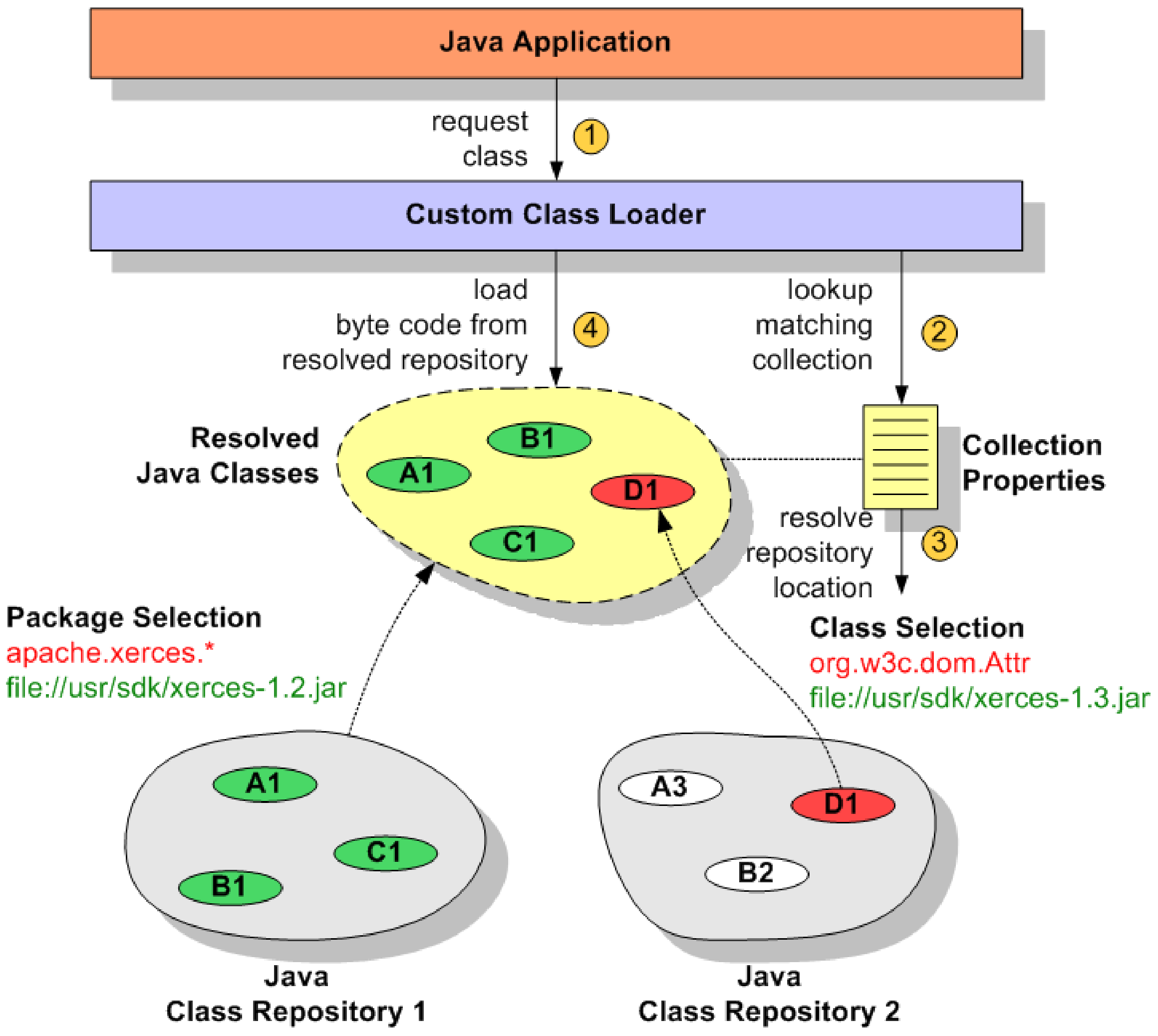


Figure 4.10: Resolving Java Classes using Java Class Collections

In the example, a new class collection is defined by selecting classes from two different class repositories. The resulting collection encloses all classes from the Java class repository 1 that match the regular expression `apache.xerces.*` and a single class `org.w3c.dom.Attr` from the Java class repository 2. In addition, the resulting class collection is decorated with the properties `name`, `version` and `creator`. They can be used later to address this class collection. If the Java application refers a class not yet loaded, a class loading request is delegated to the installed custom class loader (1). The class name is used to lookup a matching class collection (2) and to resolve the class repository location from where the associated byte code should be retrieved (3). Finally, the byte code is loaded and the requested class is instantiated by the class loader (4).

The realization is based on the introduction of a custom class loader that evaluates XML configuration file to determine the location of the byte code when a class loading request is received. An example is presented in Figure 4.11 and shows a sample configuration for the classes from the Sun Java Mail framework.

```
<collection name="javamail" id="javamail">
  <variant>
    <property name="vendor" value="sun"/>
    <property name="release" value="1.1.3"/>
    <repository>
      <location url="http://crossware.org/sun-javamail-1.1.3/mailapi.jar"/>
      <location url="http://crossware.org/sun-javamail-1.1.3/mail.jar"/>
      <package name="com/sun/mail/.*"/>
      <package name="javax/mail/.*"/>
    </repository>
  </variant>
  <variant>
    <property name="vendor" value="sun"/>
    <property name="release" value="1.2"/>
    <repository>
      <location url="http://crossware.org/jcc/sun-javamail-1.2/mailapi.jar">
      <location url="http://crossware.org/jcc/sun-javamail-1.2/mail.jar">
      <package name="com/sun/mail/.*"/>
      <package name="javax/mail/.*"/>
    </repository>
  </variant>
</collection>
```

Figure 4.11: XML Configuration Files used to define Java Class Collections

First, it declares a collection with the name `javamail` and then it specifies two variants of it decorated with the vendor tag `sun` and the release parameter `1.1.3` and `1.2,` respectively. Further, it defines the location of the related class repositories and indicates which classes from this file should be considered. For flexibility and efficiency, the declaration is done with regular expressions, thus it is easy to address various classes with few statements. As a result, class collections can be easily used by the platform administrator to define groups of classes and their host-specific file locations. Different variants of the same collection can be defined by adding additional `variant` section which eases the management of ongoing developments and concurrent implementations. Each variant can be individually marked and selected with a custom set of properties and configured to include only certain classes matching the given regular expressions. There is no need to modify existing class repositories, as shown for the Sun Java Mail classes. The XML configuration file can also be stored in a different location than the class files. This way, different platforms can also define individual collections but can still use the same class repositories, e.g. retrieved from the same URL.

4.2.2.5 Autonomic Application

The Java Class Collection approach enables an application framework to implement *Autonomic Byte Code Selection* by introducing a self-managing code repository handler, as shown in Figure 4.12.

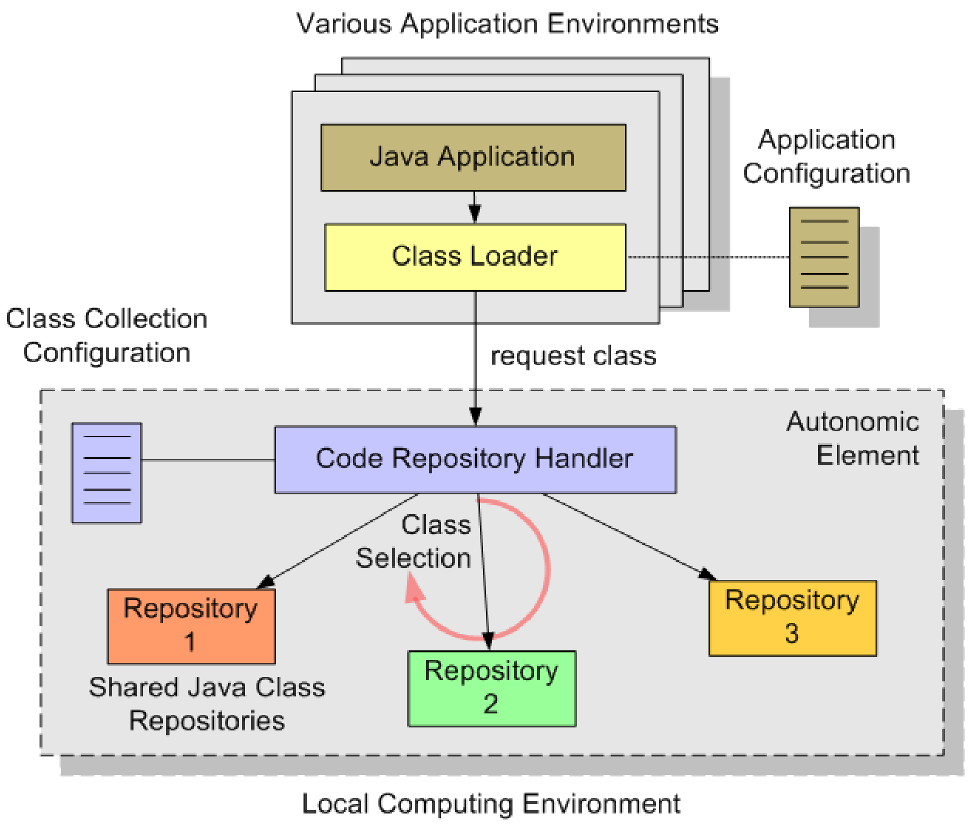

Various Application Environments
Java Application
Class Loader
Application Configuration
Class Collection Configuration
request class
Autonomic Element
Code Repository Handler
Class Selection
Repository 1
Repository 2
Repository 3
Shared Java Class Repositories
Local Computing Environment

Figure 4.12: Autonomic Byte Code Selection

Whenever an application class is to be loaded by the class loader, it delegates the fully-qualified class name (FQCN) to the code repository handler and asks for loading the related byte code from attached Java class repositories, e.g. handling JAR files located in a plugin directory. The actual resolution and loading of the byte code are hidden from the class loader that simply receives the byte code.

The autonomic operation is implemented as follows. The application configuration is used to specify the rules for selecting a class collection whereas the class collection configuration specifies the location of related byte code files. Both configurations are evaluated during runtime and set up the autonomic operation of the code repository handler. From this point of view, the class repositories are the managed elements, the class loader is the requesting element and the code repository handler acts as the autonomic manager that controls the byte code. As an example, if an application requests various Java class collections, the code repository handler resolves dependencies to other class collections in the background and considers already loaded classes to avoid class loading conflicts *(self-configuration)*. In this line, the code repository handler may check the current class loading scenario in a multi-tasking runtime environment and adjust the class loading of the next application to share and reuse already loaded classes to keep the memory foot print low *(self-optimization)*. Next, some classes may not be loaded though the related Java class collections have been successfully resolved. The resulting `ClassNotFoundException` is catched by the code repository handler and triggers the lookup of another class collection suitable to load the requested class *(self-healing)*. To prevent an application to inject malicious Java classes into a multi-application framework, the code repository handler checks all class loading requests and can deny access on class repositories following various rules, e.g. allowing only class loading from certain code repositories in the Intranet *(self-protection)*.

4.2.2.6 Related Work

The customization of Java class loading is supported and encouraged by the standard JRE implementation back to the first releases of Java. The basic principle is the introduction of a custom class loader that is used in place of or in addition to the legacy class loader. A basic categorization can be deduced along the class resolution strategy.

Native Class Assemblies. In the first category, the custom class loading implementations rely on native class assemblies based on ZIP-like data containers, e.g. JAR and WAR files. The introduced custom class loader works with the same class resolution and loading strategy as the standard Java class loader. The classes are placed in a hierarchical organization according to their packages and the resolution is directed by evaluating the package name of the requested class. In a similar sense, the class loading is performed either by reading the unzipped classes from a local file system or reading the ZIP file into memory. In a typical scenario, the required class assemblies are selected by the application developer and deployed by the application installer together with the rest of the application files, e.g. packaged on a distributable CDROM. Mostly due to potential class loading conflicts, there is no intentional sharing of the application class assemblies among various applications. For example, a servlet container like *Apache Tomcat* introduces distinct class loaders for every Java servlet and separates the application class assemblies by putting them in different folders. There is typically no support to customize the class selection during runtime, e.g. by evaluating version properties from the manifest file to resolve class loading conflicts with already loaded class instances. From this point of view, related solutions based on the original Java class loading strategy are not applicable in a multi-application scenario as addressed by the Java class collection approach and found in a cross-platform operating environment.

Shared Class Repository. The second category is related to class loader approaches that are based on a shared class repository and advanced class selection strategies. In contrast to the first category, the deployment of a class assembly is not dedicated to a single application installation. A common repository, e.g. a relational database or file system, is used to store class assemblies and related metadata information, e.g. describing the dependencies between a class and its referenced classes. If an application is to be started, a custom class loader reads the application description, connects to the shared class repository and selects the required classes by evaluating the application description and the class metadata information. Besides various proprietary approaches [155], a well-known metadata standard to describe class relationships is *OSGI* that is the foundation of many application frameworks like *Eclipse Equinox*, *Apache Felix* and *Knoplerfish* [95, 13]. In contrast to the Java Class Collection approach, OSGI metadata is typically packaged within a JAR file and refers to the contained classes only. The definition of remote class locations, e.g. referring to a database entry, is not addressed. The same is valid for remote evaluation of class properties and dependencies during runtime, e.g. for selecting matching classes without actually downloading and unzipping the JAR file. Nevertheless, the support of OSGI bundles is an interesting issue for future Java Class Collection implementations due to the increasing adoption of the OSGI standard and availability of enriched JAR files. Similar to the OSGI standard for launching Java applications, the development tools *Apache Maven* and *Apache Ivy* introduce another metadata standard for automating the dependency resolution [14]. Regular JAR files and related metadata are stored in a remote repository from where the required JAR files are dynamically loaded as they are needed during a Java build process. The developer does no longer need to resolve the application dependencies and upcoming updates of referenced JAR files are automatically included. In contrast to OSGI, Apache Maven supports the remote evaluation of class assemblies. There are various features to manage local bundle caches to avoid repeated download of the same class assemblies. By focussing on the resolution of class assemblies and Java build automation, there is no way to define subsets of classes based on existing class assemblies like in the Java Class Collection approach.

Networked Class Loading. While class loader implementations of the categories above rely on locally stored class assemblies, the third category is dedicated to networked class loading where required classes are dynamically resolved and loaded from remote class repositories. A well-known approach is *Java Network Launch Protocol (JNLP)* and its reference implementation *Java Web Start* from Sun [383]. Based on regular JAR files and an application description file, a custom class loader typically retrieves related class assemblies from a web server via HTTP. This results in the main advantage of the JNLP approach, namely the on-demand selection and downloading of application classes via the Internet. The JNLP specification allows developers to define custom class assembly configurations that are selected during runtime according to the target runtime environment and operating system. In comparison to the Java Class Collection approach, however, the JNLP assembly definition is fixed. There is no option to adjust the class configuration of an application and there is no support to dynamically resolve class dependencies across different JNLP sites. Besides further remote class loading approaches like the use of a *remote Java codebase* in RMI applications, there are countless custom network class loaders. They typically add the ability to download classes over different communication paths, e.g. by using regular HTTP servers as in *Harness* [209], *JXTA pipes* to transmit class files in a P2P network [288], a *Java Class Broker* to retrieve class instances from a remote peer via a TCP socket connection [164] or by extending an application server with customer services loaded from an HTTP server [265]. In general, the legacy Java class loader extensions do not change the way how classes are selected but how specified classes are retrieved from a remote site. In contrast to that, various ideas have inspired to change the class selection according to application and environment requests. In *Gandiva* [227] for example, Java classes are dynamically determined and selected during runtime, e.g. according to the security constraints in a distributed application environment. A remote class inventory is introduced that can be queried for matching classes by evaluating their custom metadata definitions.

In summary, the legacy Java class loader approach encourages the custom adaptation how classes are loaded but not how they are selected. While there are many custom class loader implementations for various purposes, only few allow changing the class resolution and selection strategy. A related specification standard for dependency configuration is missing though OSGI has received much attraction in recent years and addresses some objectives of the Java Class Collection approach such as the custom assembly tagging. Nevertheless, the Java Class Collection approach is not bound to a certain application framework or based on the introduction of a custom programming model. It can be used with legacy Java applications and easily integrated in existing application frameworks.

4.3 Dynamic Software Composition

Software applications are usually developed by reusing existing code components and implementing new features specific to the current business logic. Besides static software composition, e.g. by linking libraries during design time, dynamic software composition is performed during runtime and using a composition configuration defined by the application assembler. When the application is loaded, the runtime system evaluates the composition configuration and tries to resolve all required component assemblies, e.g. by looking for shared libraries using the given name. This approach works well in a single application scenario with only one composition configuration in action, e.g. running applications in separate address spaces. To run multi applications in a Java Virtual Machine (JVM), composition configurations are typically used to build specific class loader hierarchies for shielding different and sharing common classes among applications. However, conventional approaches may not dynamically change the composition strategy, e.g. by reusing a compatible code assembly from the local cache instead of downloading the requested variant from a remote code repository. In Section 4.3.1, a new class loading approach called *Java Class Spaces* is presented that enables the fine-grained adaptation of application composition during runtime by evaluating the current composition setting, application composition requests and a system composition policy [267]. Another issue of Java application composition is the provision, request and sharing of random program resources like database connections and user profiles. In Section 4.3.2, *Java Loadable Modules* are introduced that represent initializable code components and manage various program resources [272, 360]. A related module manager receives module resource requests from the application and performs the lookup, the dependency resolution and the initialization of suitable modules in a cross-platform operating environment.

4.3.1 Java Class Spaces

The self-managing organization of Java class loaders using Java class spaces and configuration policies is presented in the following section. It is based on application assembly definitions and code packages [269], as introduced in Section 4.2. Afterwards, the particular handling of class loading requests for assigning Java classes to distinct class loaders is described. Finally, the self-managing adaptation of application composition in a shared application hosting environment is illustrated.

4.3.1.1 Motivation

The regular Java Virtual Machine (JVM) uses the system class loader to load and resolve application classes. While this approach is feasible for the original single application hosting approach, advanced application hosting systems transform the JVM into a multi-application hosting environment by instantiating various custom class loaders; usually one per application as in a typical Java servlet engine implementation or application server configuration. All class loaders are chained in a tree-like hierarchy where every class loader has one parent class loader and may have several children class loaders. At top of the hierarchy, the boot class loader and its sole children, the system class loader, are instantiated by the JVM. If an application issues a class loading request, the related application class loader checks if the class has been already loaded and returns the instantiated class. If not, it delegates the request to its parent class loader that performs the same check and, if necessary, passes the request further to its parent. This is repeated until the root of the class loader hierarchy, the boot class loader, is reached. Then, the boot class loader tries to locate the class and if it is not found, the request is passed back the same way down until a class loader on the path can load the class. Finally, if the issuing class loader can also not load the requested class, a `ClassNotFoundException` is raised.

The standard class loading procedure ensures that first the parent class loaders get the chance to load common classes that are visible to all children class loaders and loaded only once, e.g. Java system classes like `java.lang.String`. In turn, classes loaded by a children class loader are not visible to the parent or the sibling class loaders; hence shielding private application classes from other application instances, e.g. concurrently loaded classes of different application versions. However, a basic problem is the concrete specification which class loader should be actually used to load the class. As described above, the first parent class loader that is able to locate the requested class will load it regardless if the application wants to use a private class variant. In addition, the regular class loader implementation uses only the package and class name to locate a class. For the resolution of related class dependencies, the system cannot determine which class loader has already loaded the application class and should also be used to resolve the dependencies.

4.3.1.2 Features

The goal of the *Java Class Spaces* approach is the introduction of a self-managing Java application composition framework with particular respect to multi-application hosting in a regular JVM. The major features are as follows.

Java Application Definition. The JCS approach allows developers to specify the application composition without knowing the deployment configuration on the target platform. A developer can define class collection queries that are evaluated by a custom class loader and used to retrieve suitable Java class collections on-the-fly. If a composition request cannot be fulfilled, another component can be selected without running into an unexpected `ClassNotFoundException`.

Adaptive Composition Strategy. With JCS, the application composition configuration does not request specific component assemblies but describe the properties of a matching Java Class Collection. This way, a self-managing class loader is able to refine the application composition by considering the current composition layout, cached component assemblies and deployment efforts for retrieving new assemblies from remote code repositories, e.g. in case of a low-bandwidth network link.

Shared Application Environment. A particular benefit of the JCS approach is the ability to organize and manage Java classes from various applications in a shared application environment. Every application is associated with a class space that in turn is managed by a custom class loader. A self-managing class space supervisor takes care about the overall organization and decides about sharing and shielding of application classes according to the application composition strategies.

Dynamic Class Reloading. A special issue in self-managing application composition is the reconfiguration of an existing composition layout during runtime, e.g. upgrading to a newer component version or replacing a buggy class instance. To this end, the JCS approach allows virtually reloading of Java classes by recording the current configuration, removing the affected class space and inserting a new one without user intervention.

Legacy Java Programming Model. The JCS approach enables adaptive application composition without introducing a different Java programming model. Any legacy Java application can be loaded and run as long as they do not install another class loader; hence they rely on the standard class loading procedure. In particular, existing Java components and libraries can be used for self-managing application composition after being packaged using Java Class Collections.

4.3.1.3 Approach

The *Java Class Space* approach enables the customizable organization of Java classes using a configurable hierarchy of class loaders. Every class loader is associated with a class space and adjust the delegation of class loading requests to the parent class loader according to a given class space configuration. Related classes may be grouped in the same class space while other classes are put in a separated class space, e.g. shielding the Java classes of concurrently hosted Java applications in the same JVM, as shown in Figure 4.13.

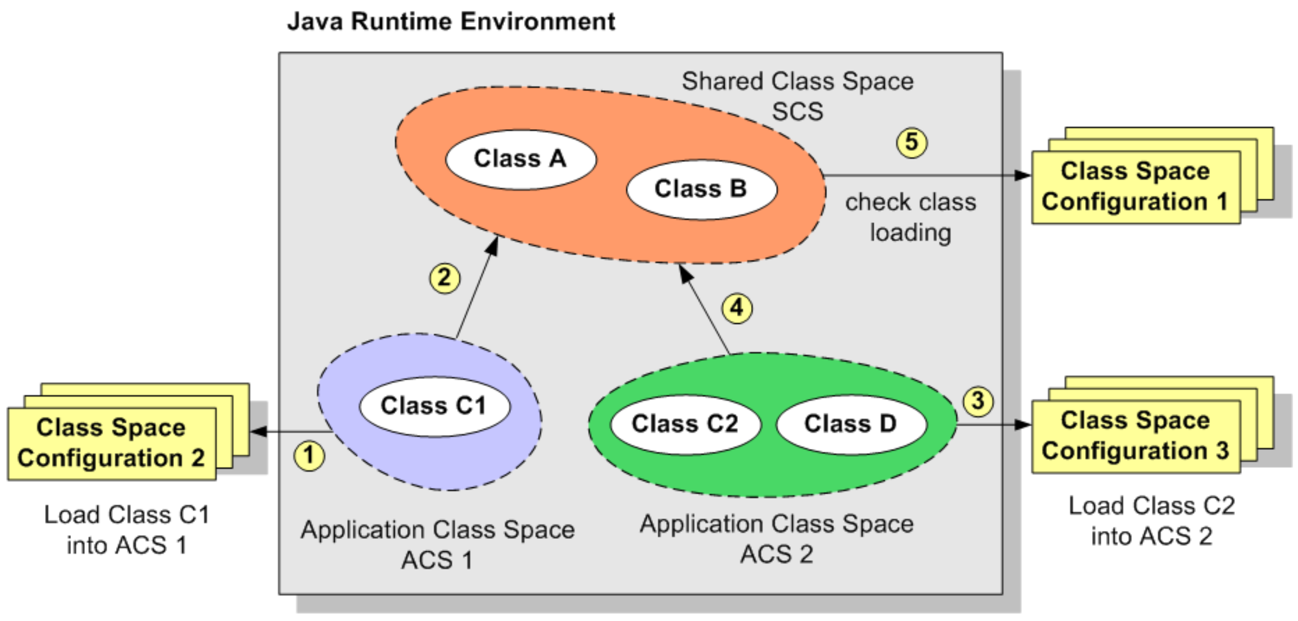


Figure 4.13: Self-Organizing of Application Code using Java Class Spaces

There is a shared class space SCS and two application class spaces ACS1 and ACS2 as children of SCS. All objects created by classes in ACS1 will be associated with the class loader of ACS1. In the case one of these objects creates another object from a new, yet not loaded class, the class loader is asked to load the byte code of the newly requested class. For this purpose, it determines whether the related class space, in this case ACS1, is configured to load the class. If not, it delegates the request up to its parent class space SCS and its class loader. This is directly opposed to the original behavior of chained class loaders, where all class loading requests are at first directed to the parent class loader. But with this reversal, the class spaces can be configured to hold shared classes in the parent class spaces and unshared classes in the child class spaces. In the example, objects in ACS1 and ACS2 can be built from classes with the same name C, but could rely on different class implementations. On the other hand, they can share data and collaborate by using classes and objects located in SCS. In addition, the configuration of a class space and the information which classes can be loaded is combined with the information where to find the related byte code. A special issue at this point is the question of how the system should resolve registration conflicts between parent and child class space, if both want to register the same class for loading. The answer is quite simple: all registrations must be checked against the configuration of the chained class spaces at startup, and if there is already a chained class space that handles the class, the registration is denied. This is manageable up to a certain degree if the class spaces are configured properly by the application framework. But certainly, there are constellations that cannot always be resolved, i.e. applications, which want to collaborate using the same class in different variants. In this case, however, some kind of adaptation is needed anyway.

As a result, the application code layout, shared and separated classes is defined without modifying existing application code. The configuration process of class spaces and the distribution of newly loaded classes are completely transparent to the application. Of course, this is only guaranteed as long as the application does not use a self-defined class loader. In this case, it would be possible, that the foreign class loader retrieves classes without knowledge of the parent class loaders and their associated class spaces. Thus, they would not be able to check future resource registrations whether the related classes are already handled by one of the child class loader.

4.3.1.4 Realization

Before an application is started in a JVM, an application class loader is created and associated with an application class space, as shown in Figure 4.14.

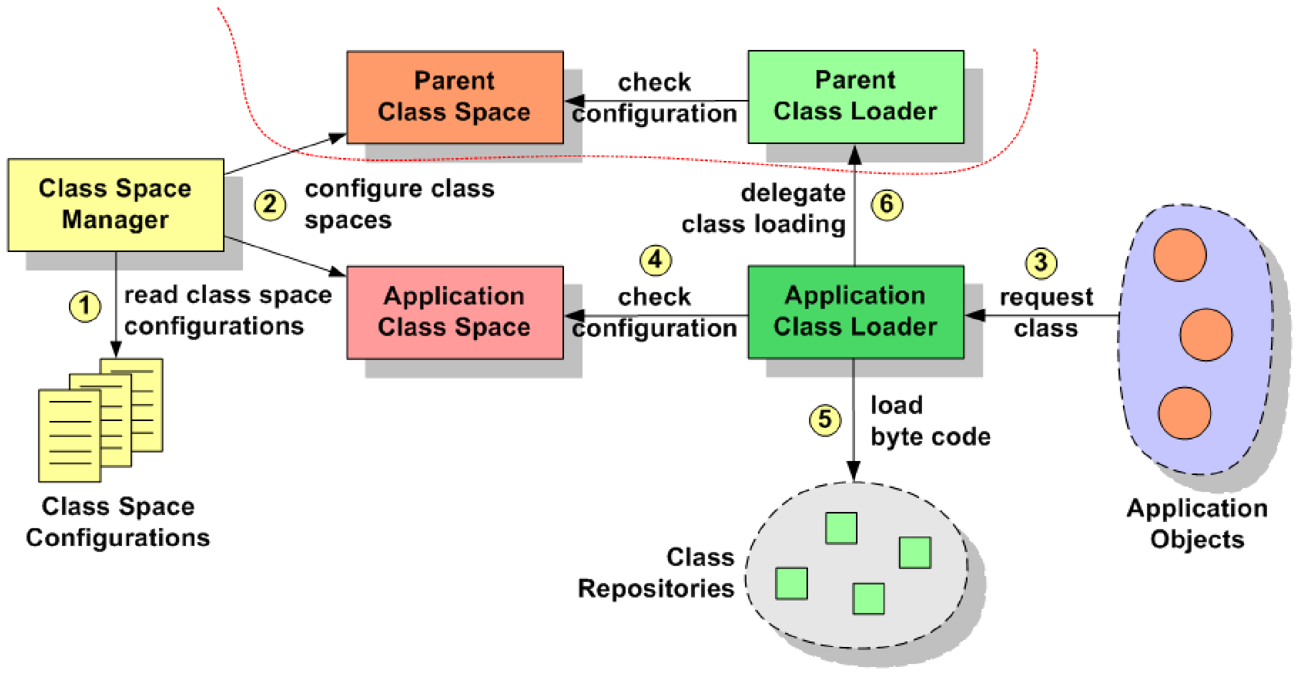


Figure 4.14: Application Class Loader and Application Class Space

The class space is initialized with a class space configuration file and evaluates matching class collection configurations to determine which classes can be loaded and which class loading request should be delegated to the parent class space. After that, the first application class is loaded using the application class loader and thereby associated with it. As a result, subsequent requests of further application classes are automatically handled by the same application class loader.

Dynamic Class Space Configuration. The configuration of class spaces can be done statically by using an XML configuration file given by an administrator, or dynamically by the application implemented by a developer, as shown in the example in Figure 4.15.

```
IClassManager mgr = ...;
IClassSpace app = null;

app = mgr.createClassSpace("application", "system");

app.registerResource ("org/apache/xerces/.*",
                      "/usr/lib/xerces-1.4.4/xerces.jar");

ClassLoader loader = app.getClassLoader();
Class parser = loader.loadClass("org.apache.xerces.parsers.DOMParser");
```

Figure 4.15: Example of Dynamic Class Space Configuration

At first, a new class space *app* is created with the class space *system* as its parent. Then, a new resource is registered in the class space, specifying that all classes whose names begin with *org.apache.xerces* will be found in the given JAR-file, and the class space should be able to load this class. By the way, the dots in the class name are masked with slashes, since a dot is a special letter in regular expressions, which are used to specify the matching pattern. At the end of the example, the class space is requested to load a class, or with other words, the class is injected into the class space. All subsequently class loading requests initiated by this class, respectively its objects, will also be handled by the class loader of the associated class space *application.* At this point, it should be stressed that the class space does not load any class without a request. The configuration does only specify which classes can be loaded by the class space and where to find the related byte code.

Static Class Space Configuration. Besides dynamic class space configuration that is primarily used by developers, an administrator can define a static class space configuration for the JVM. It is read when the system is started and enables administrators to specify the class space of each application separately. An excerpt of an example is shown in Figure 4.16.

The configuration defines a class space called *application*, with the class space *system* as its parent. The class space *application* is configured to host classes from the JAR-files specified in *path* and the following *resource* entries. The class space *application* is not automatically created, but whenever the system opens a class space named *application*, the configuration file is read, and the class space is configured respectively. Afterwards, additional resources can be registered dynamically provided that they do not cause any conflicts with existing registrations.

```
<space name="application" parent="system">

 <jar path="/usr/sdk/servlet.jar">
   <resource name="javax/servlet/.*" />
 </jar>

 <jar path="/usr/sdk/xalan.jar">
   <resource name="apache/xalan/.*" />
 </jar>

</space>
```

Figure 4.16: Example of Static Class Space Configuration

4.3.1.5 Autonomic Application

The Java Class Spaces approach enables the *Autonomic Class Organization* for the concurrent execution of Java applications by introducing self-managing class space handlers, as shown in Figure 4.17.

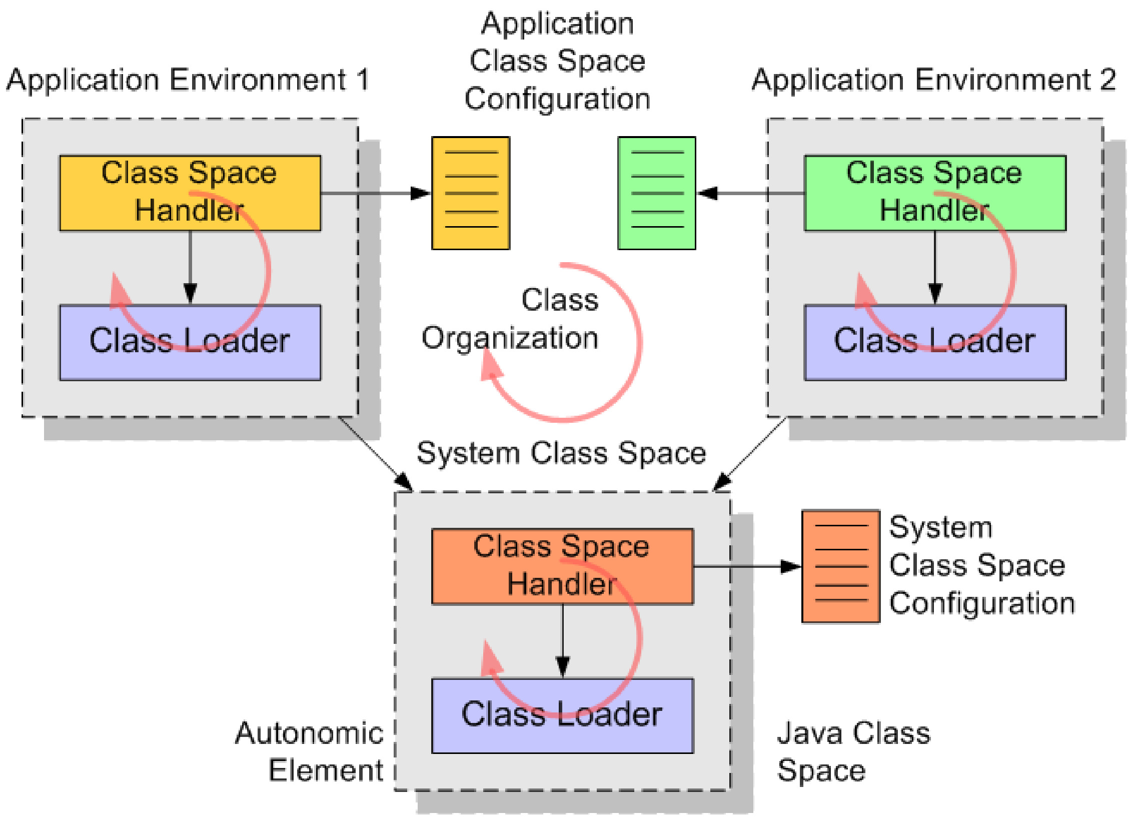


Figure 4.17: Autonomic Class Organization

If a Java application is launched, a new class loader is created by the global class space manager and associated with a class space as part of a hierarchical class space system. A class space handler monitors the class loader and controls the way how classes are organized across the class spaces.

Every class space is configured with a class space configuration that contains class loading rules, e.g. to delegate the loading of certain classes to a parent class space or to prevent an application class space to reload a private copy of a shared class instance. The processing of class loading requests is performed in a self-managing way by evaluating the class space configurations and by monitoring the class loading requests during runtime. From this point of view, the class space represents the autonomic elements, the class space handler the autonomic manager and the class loader is the managed element. In particular, the ever-changing loading and launching of applications in a multi-tasking runtime environment requires a flexible management of shared and private class instances. For example, if an application environment for running a new application is set up, the class space manager determines which Java class collections may be shared with all or some applications. This results in an adaptive organization of class spaces with respect to the current class spaces layout *(self-configuration)*. During runtime, a loaded class instance may have to be replaced with a different one, e.g. due to a hot upgrade. In this case, the implementation supports the unlinking of a class space branch and the insertion of reconfigured class spaces *(self-optimization)*. The affected applications are typically relaunched after that or suspended and continued using the object spaces approach, described in Section 4.6.2. This is also the case if the class space handler detects a class loading conflict due to a configuration error and has to reconfigure the related class spaces *(self-healing)*. Finally, the class spaces implementation is also used to shield private classes of an application against the unauthorized access of concurrently executed applications hosted in different class spaces. This is ensured by the class space handler that monitors every class loading request and denies access to objects not belonging to the same application class space *(self-protection)*.

4.3.1.6 Related Work

The introduction of custom class loaders as proposed for implementing the Java Class Space approach is a widely used strategy. In a single application scenario, the custom class loader is typically added to extend the loader capabilities of the system class loader, e.g. for loading classes from a remote site. In a shared application scenario, various class loaders are concurrently instantiated and controlled by an application framework to address the particular needs of multi-application hosting like the sharing of common classes. In the following sections, related work is considered that allows defining and managing the class loader organization during design or runtime, respectively.

Single Application Environment. The regular Java Runtime Environment has been originally designed for single application execution and locally stored class assemblies. This results in a simple class loader configuration that does not offer advanced definition of runtime requirements and composition schemes, e.g. for separating the tasks of application developer and assembly deployer [see Section 2.3.3]. A well-known solution to that is provided by the *Java Network Launch Protocol (JNLP)* and its reference implementation *Java Web Start* from Sun [383]. An application configuration file is introduced to define application properties like the required operating system to run the application and remote locations where to find the application classes. Similar to the Java Class Space approach, the JNLP approach also supports the legacy Java programming model and the use of regular class assemblies. If an application is to be launched, a particular JRE is started and a custom class loader is installed before the actual application is loaded. From this point of view, the JNLP approach represents a standardized implementation variant of the Java Class Space approach though there is no support to enable runtime adaptation of the application composition, e.g. due to user request. A related *runtime adaptation approach* is presented in [266] where web server functionality is dynamically extended on user request by downloading code assemblies from the Internet. In contrast to the Java Web Start approach, there is no fixed application definition that can be checked and only one class loader is used. Thus, no concurrent hosting of different class variants is possible and the reloading of classes is not supported. These issues are addressed by a proposal of *hierarchical arrangement of custom class loaders* in [329]. An application deployment descriptor is introduced that can be used to declare the class loader organization, similar to the Java Class Spaces approach. Every application receives its own class loader and application components may be put in extra child class loaders, e.g. concurrently loaded application plugins. As a result, selected application classes are separated from each other and may be reloaded without affecting concurrently loaded components. In contrast to the Java Class Spaces approach, the application deployment descriptor is not able to select certain classes from a given class assembly, as shown in Figure 4.16, and there is no dynamic configuration option that can be applied during runtime. A more advanced class loader organizing approach is presented in [42] and which introduces a *token-driven class loader mechanism*. This is used to reverse the class loading delegation from a parent class loader to one of its child class loader, e.g. in an application framework scenario where a requested plugin class does not have to be loaded by the framework but by the delegated plugin class loader. This is very similar to the Java Class Spaces approach but a token is used to select a matching class loader instead of regular expressions for filtering the handled classes.

Shared Application Environment. With the advent of application frameworks, the demand to host multiple applications within the same JVM has brought up new class loading issues, e.g. handling complex dependency relations between shared framework classes and specific classes of isolated application instances. In general, the class loader approaches designed for single application environments may be applied and extended in a shared application environment, e.g. to organize class loaders in a hierarchical manner [139]. A new issue is the possible occurrence of class loading conflicts due to different class variants requested by multiple applications not known in advance. Popular application servers like *Jakarta Tomcat*, *JBoss* and *Jonas* [16, 179, 187] instantiate a separate class loader for every loaded application instance and hereby strictly separate application-related classes from each other. Apart from putting shared classes into the framework CLASSPATH, the configurable organization of shared and shielded classes as in the Java Class Space approach is not supported. Application frameworks, e.g. based on *OSGI* [262], allow reloading of application instances during runtime due to software updates. In this case, an existing class loader is replaced with a new one, however, without re-creating the original class loader hierarchy, e.g. for reloading application plugins as well. A transparent use with legacy Java applications as possible with the Java Class Space approach is not feasible. A different approach to organize the Java class loading across various class loaders is described in [81]. It is based on the *Multi-Tasking Virtual Machine (MVM)* [77] and allows sharing the runtime representation of classes across concurrently hosted *application isolates* within the same JRE and across distinct MVM instances. A similar extension has been added to the JRE with *class data sharing* in Java 5.0 [337]. From a multi-application hosting point of view, these approaches reduce the overall memory footprint and startup time of an application if already loaded classes may be reused. The class loader organization, however, cannot be managed by the application environment itself, e.g. to replace already loaded classes, but it is part of the underlying JVM implementation. As a consequence, the custom class loader implementations can be usually found in application frameworks and runtime environments that rely on particular programming models, e.g. Java servlets, OSGI services and EJB component model.

In summary, there are different approaches of how applications can be hosted and related classes are organized. For single application environments, the JNLP approach represents a good choice to define application runtime requirements and composition schemes. Most of the Java Class Space features are supported and Sun's Java Web Start is usually installed as part of a regular JRE. Though the Java Class Spaces approach is a proprietary solution, it can be easily integrated in existing applications, e.g. to enable adaptive Java application composition. For shared application environments as typically found in an on-demand computing scenario, the Java Class Space approach encompasses the custom class loader features of established application frameworks and adds new ones, as the dynamic recording and restoration of hierarchical Java Class Space configurations. A common problem of existing class organization solutions is the lack of instrumentalization support, e.g. by an autonomic class loader like in the Java Class Space implementation.

4.3.2 Java Loadable Modules

The regular Java Runtime Environment (JRE) lacks native support of a dynamic component model and the lookup and deployment of distributed component assemblies. The *Java Loadable Module* approach introduces a module-based component and deployment model, and supports the managed access to contained resources in a multi-application environment. The runtime adaptation of the application composition is enabled by transparent module rebinding and individual life cycle management. In the following sections, the Java realization is described and its application for the self-integration of dynamically loaded modules and instantiated resources is shown.

4.3.2.1 Motivation

The dynamic composition of software applications has several advantages regarding the on-demand extension and adaptation of application systems [5, 18]. In contrast to a self-contained application assembly that is loaded as a whole before the related application is started, various software components are loaded and put together one-by-one as they are needed [123]. This reduces the startup time of the application and enables the adaptation of the program functionality during runtime, e.g. by selecting and loading distinct components according to a plugin configuration. In a shared application environment, common components like an XML parser may be shared by concurrently hosted applications to reduce the overall memory usage. In addition, already loaded software components may be individually replaced by different variants on-the-fly without shutting down the host system along with concurrently running applications, e.g. while updating a certain plugin [27].

For better handling, related software components are usually grouped in *component packages* that are loaded under particular control of the runtime environment, e.g. for resolving the component dependencies, managing the life-cycle and handling security issues. Well-known component packages for dynamic application composition are Dynamic Link Libraries (DLL) for MS Windows applications and Shared Libraries for Linux applications. For Java applications, a Java Archive (JAR) seems to be a comparable component package. However, the JAR file only groups various classes in a ZIP file for better deployment and does not act as a component package that can be addressed by an application. Apart from loading single classes by their name, there is no common support for life cycle management of arbitrary program elements provided by a module, e.g. a database handler or a window manager in a shared application environment. In addition, the support of dynamic rebinding updated component instances is another upcoming issue of distributed operating environments [21].

4.3.2.2 Features

The overall goal of *Java Loadable Modules (JLM)* is the provision of application component units which separates the virtual application composition from the physical component deployment. The major features are as follows.

Application Component Units. With the JLM approach, initializable application component units are introduced that represent functional blocks that can be explicitly requested by an application. The component developer can group various components into a module and may add a module handler to initialize contained resources on first access. A module manager is introduced that controls module handling and provides seamless access to loaded module instances.

Individual Module Resolution. While every JLM is associated with a unique module identifier, an accompanying module configuration contains extra property definitions and dependency declarations that can be evaluated during runtime. In turn, an application developer can specify module loading requests during design time, e.g. querying modules offering a specific resource, without knowing how to load matching modules from a local or remote module repository.

Life Cycle Management. While in a shared application environment a JLM can be concurrently requested by one or more applications, the module manager ensures that every module is loaded only once; eventually reusing an already initialized instance. If an application does no longer need a module, it is released and the module manager checks if it can be removed. As a result, the life cycle of a module instance is controlled by the module manager which eases the application development.

Self-Managing Module Binding. The module manager tracks the loading of modules and controls their proper operation. In case a module cannot be initialized, indicates an error later on or should be upgraded, the module manager substitutes the related module by loading and initializing another module variant. In this context, existing module resource bindings are seamlessly restored without notifying the application; hence introducing self-healing features to the overall system.

General Composition Approach. The module approach is not limited to Java applications. There are also C++ implementations for MS Windows and Linux which, in particular, unifies the binding of static and shared libraries and separates their different handling from the application implementation. Instead, the module manager detects the type of binding during runtime and provides seamless access on module resources on-the-fly.

4.3.2.3 Approach

In regular Java application environments, the application composition is triggered by explicit loading of specific code assemblies and by requesting needed resources afterwards. The basic idea of the following approach is the reversion of this process and the separation of application composition and assembly loading. *Java Loadable Modules* are introduced that dynamically select and load code assemblies due to application resource requests and by transparently evaluating the current deployment scenario, as shown in Figure 4.18.

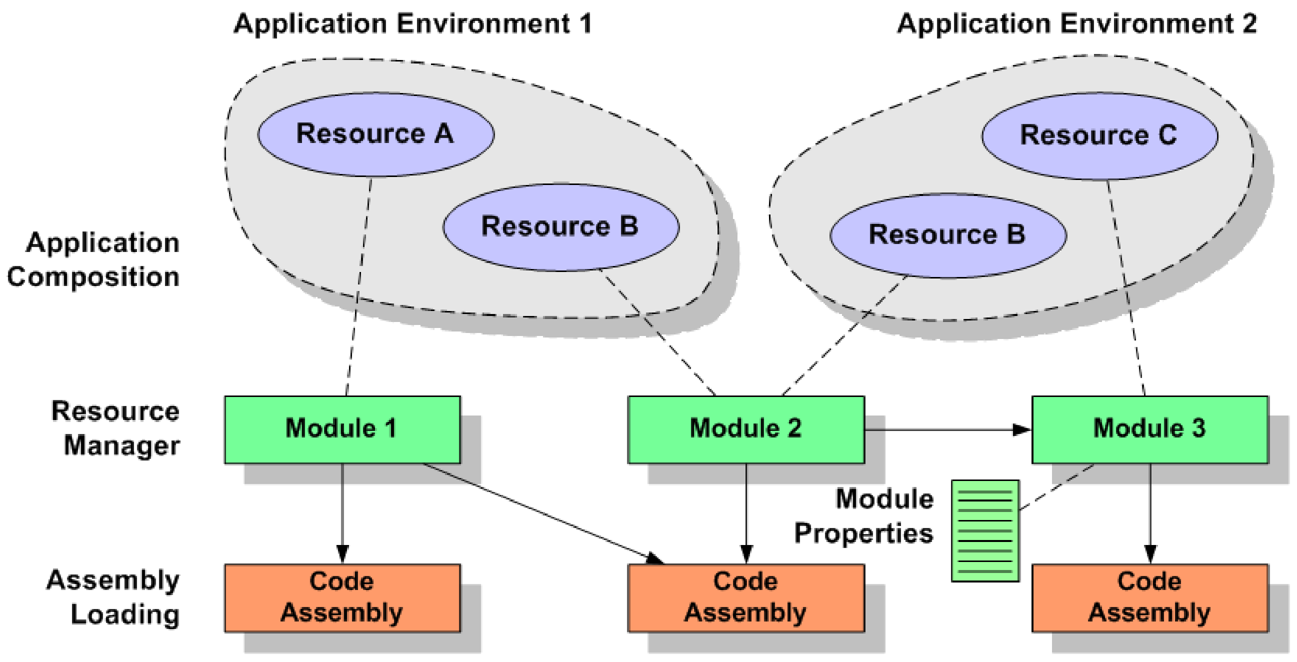


Figure 4.18: Application Composition using Modules

Similar to Shared Libraries in UNIX and Dynamic Link Libraries in MS Windows, a module represents a building block of a dynamically composed application. In contrast to a Java class collection [see Section 4.2.2], a module additionally contains a particular module handler that is responsible for initializing the module and for providing an interface to its resources, e.g. a reference to a database connection. An application is no longer requesting resources by directly loading code assemblies or using code references, e.g. a Java class name, but by querying modules containing the required resource. To this end, a module is decorated with module properties and a list of managed resources each associated with a unique resource id. Furthermore, a module is usually loaded only once when the first resource is requested. Later on, the same module is reused that supports the customizable application composition in a shared application environment. As an example, both application environments in Figure 4.18 request the same resource B that is provided by module 2 and initialized only once, e.g. a reference to a sidebar window on a graphical desktop.

4.3.2.4 Realization

The realization of the JLM approach is based on the introduction of a module manager and module handlers, as shown in Figure 4.19.

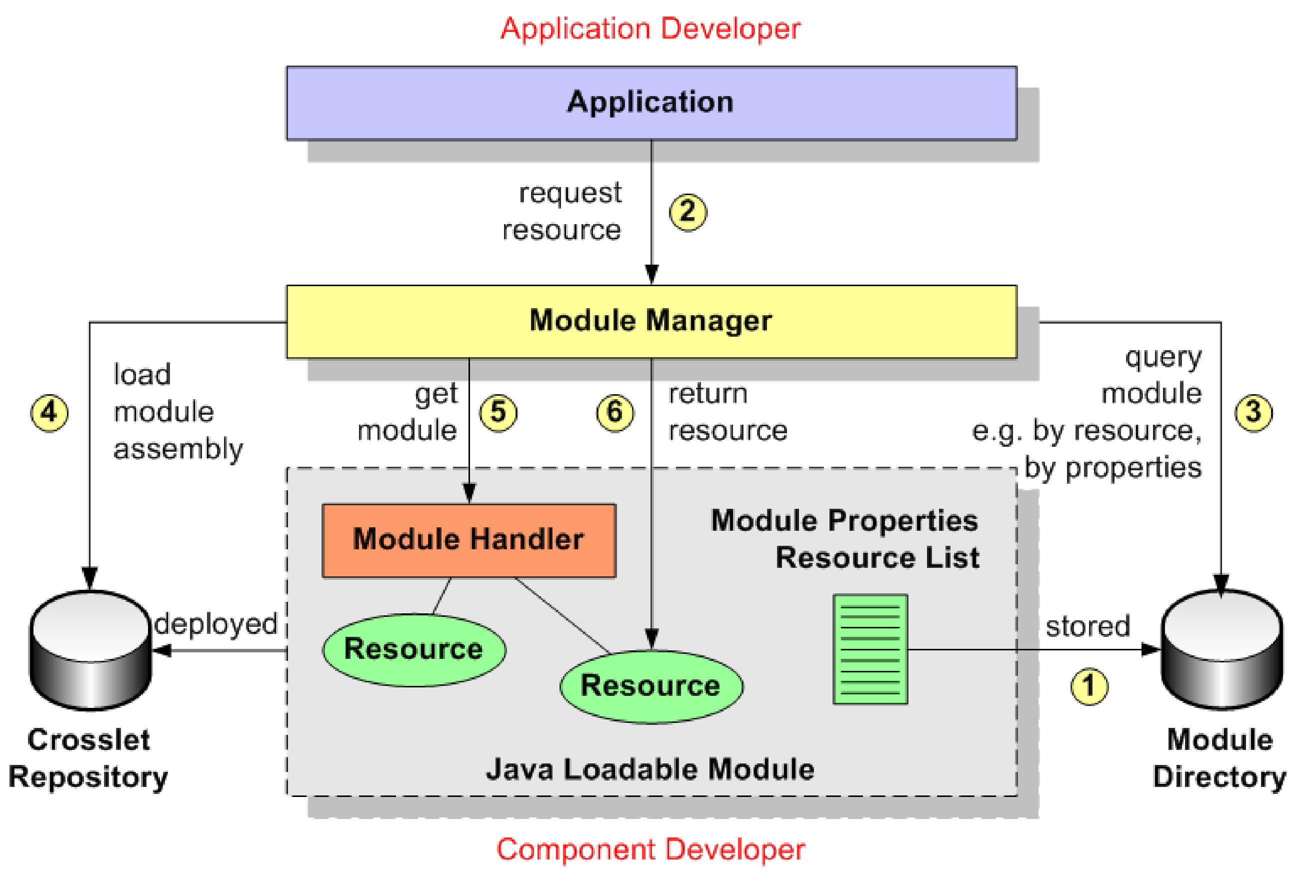


Figure 4.19: Java Loadable Module

When a computing system is set up, the system administrator configures a module directory where the module properties and resource list of all available modules are stored (1). This could be a local configuration file or a remote web service that is shared among computing systems in the same Intranet. If an application request a resource (1), the module manager, a singleton object, queries the module directory for suitable modules (2), e.g. matching a property query or offering a specific resource. Next, the module code assemblies are resolved using the crosslet repository (3) and the module handler is created (4). After having initialized all contained resources, the requested resource is returned to the application (5). In case an already loaded module contains a requested resource, the module manager skips loading another module instance and returns the loaded resource instance.

Module Handler. To use the JLM approach, the component developer has to implement a module handler and associate the module and managed resources with unique ids, as shown in Figure 4.20.

```
public interface IDatabaseModule
{
  // unique module id
  public final static CModuleId MODULE_ID = "{6A53F7E2}";

  // resource id
  public final static CResourceId DATABASE_RESOURCE_ID = "{D5378CA1}";
}

public class CDatabaseModule implements IDatabaseModule
{
  private IModule providerModule;
  private IProvider provider;
  private IDatabaseConnection conn;

  // initializing the module
  public void init() throws XModule
  {
    providerModule = openModule(IPostgresqlModule.MODULE_ID);
    provider = providerModule.openResource(IPostgresqlModule.PROVIDER);

    conn = openDatabase(provider, "address");
    registerResource(DATABASE_RESOURCE_ID, conn);
  }

  // clean-up module resources
  public void exit() throws XModule
  {
    unregisterResource(DATABASE_RESOURCE_ID);
    closeDatabase(conn);

    providerModule.closeResource(provider);
    closeModule(providerModule);
  }
}
```

Figure 4.20: Implementation of a Module Handler

First, an interface `IDatabaseModule` is defined that may be used by other parties to interact with this module. It contains a unique identifier `MODULE_ID` that tags this module and a constant `DATABASE_RESOURCE_ID` used to access a managed resource of the module. Second, there is an implementation of this interface representing the actual program code of the module handler. Its `init` method is automatically called when the module is loaded, and the corresponding `exit` method is called when the module is no longer referenced by other modules. In the example, the `init` method dynamically opens another module `IPostgresqlModule` using the related module id `IPostgresqlModule.MODULE_ID` and requests a contained resource `PROVIDER` in it. Subsequently, the actual initialization of the module is performed by opening a database connection using the requested provider object and by registering the connection as a new resource `DATABASE_RESOURCE_ID`. This resource can be requested by other modules in a similar way like the provider resource of the module before. In turn, when the module is no longer referenced, the `exit` method is called and the initialized resources as well as the requested modules are released.

Module Configuration. As shown in the example in Figure 4.20, each module is referenced by its module id without any knowledge how and where to retrieve the related module code. The module management completely hides the issues of lookup, loading and initializing modules from the application. Instead, these tasks are performed by the module controller of the application environment that is configured using certain module definition files, as shown in Figure 4.21.

```
<module id="{A9D52EF1}" handler="de.fhg.texteditor.CModule">

  <properties>
    <property name="name" value="texteditor" />
    <property name="vendor" value="Fraunhofer" />
  </properties>

  <resources>
    <resource id="{A42BA647}" />
    <resource id="{6FA53EE6}" />
  </resources>

  <dependencies>
    <module id="{2E6210AA}"/>
    <module id="{D181334A}"/>

    <collection id="texteditor">
      <property name="release" value="1.0.0" />
    </collection>
    <collection id="apache-xerces">
      <property name="release" value="2.4.0" />
    </collection>
  </dependency>

</module>
```

Figure 4.21: Module Configuration File

The module `texteditor` is marked with the module id `{A9D52EF1}` and defined to use the class collections `texteditor` and `apache-xerces` with the given properties. The attribute `handler` points to a class that represents the module handler of the current module. Furthermore, the module may also be decorated with properties like `vendor` that can be used to query this module. The section `resources` contains a list of resource descriptions that can be queried by the module manager when searching for a requested resource, e.g. by resource id. Finally, the section `dependency` indicates further modules that have to be loaded by the module controller before the current module can be used.

Component Framework. A particular module resource is a *component class* that is a template for creating a customizable component object. A component model specifies rules for defining a component class, e.g. which interfaces have to be implemented for interacting with the component container. In addition to common component interfaces, a component class may be decorated with custom interfaces that are determined during runtime and represent additionally implemented features of the component, e.g. a thread component that exposes an interface to suspend and resume the managed thread. In the implementation of the Java Loadable Module approach, a *reflective component framework* is introduced to dynamically inspect the component features, to initialize the component object and to embed it into a runtime context, as shown in Figure 4.22.

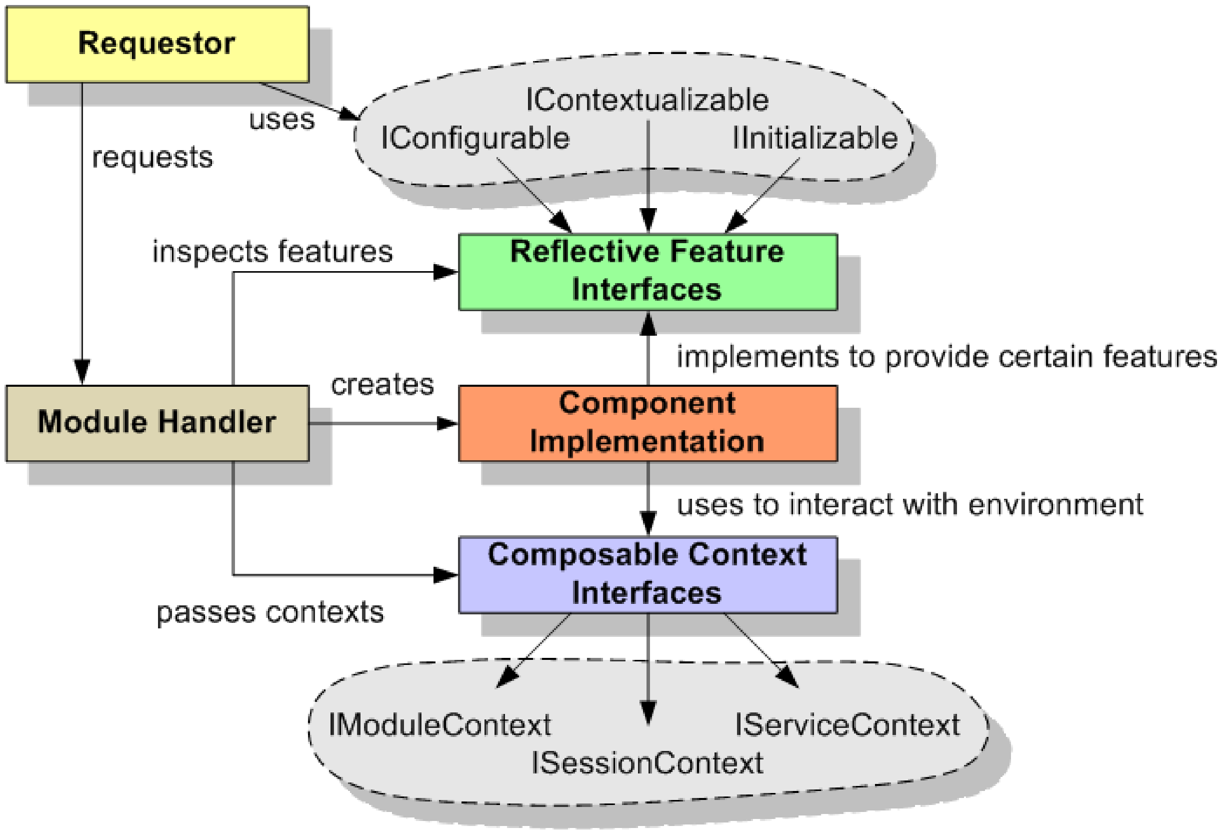


Figure 4.22: Reflective Component Framework

Every component class may be decorated with *feature interfaces* that represent a certain capability of the component like `IConfigurable`, `IContextualizable` and `IInitializable`. Following the *Inversion-of-Control pattern (IoC)* [352] they are determined during runtime by Java reflection and used to interact with the component, e.g. passing configuration parameters and object references to the runtime contexts like `IModuleContext`, `ISessionContext` and `IServiceContext`. An excerpt of a component implementation is shown in Figure 4.23.

```
public class CClassManager implements IClassManager, IConfigurable,
                                          IContextualizable, IInitializable
{
  private ClassLoader m_loader;
  private URL m_urlReg = null;
  private IClassRegistry m_reg = null;

  public void config(IConfiguration cfg)
  {
     if (cfg instanceof IClassManager.IConfig)
        urlReg = ((IClassManager.IConfig)cfg).getRegistryURL();
  }

  public void contextualize(IContext ctxt)
  {
     if (ctxt instanceof IClassContext)
        m_loader = (IClassContext) ctx.getClassLoader();
  }

  public void init() throws XInitializable
  {
     m_reg = urlReg.openConnection();
  }
}
```

Figure 4.23: Excerpt of a Component Class

A component that implements `IConfiguration` provides a method `config` that can be called to pass configuration parameters to the component before it is initialized. In a typical scenario they are read from a configuration file. A particular feature interface is `IContextualizable`, indicating a component that requires certain interfaces to access the context it has been created in, such as `IClassContext`. It typically contains object references to other components and resources created during runtime. The feature interface `IInitializable` indicates that the component can be initialized. To create a component object, a typical initialization sequence is to determine and call the methods `config`, `contextualize` and `initialize`. The requestor of a component object is not aware of this sequence but passes the related information to the component framework, as shown in Figure 4.24.

```
IComponentFactory fac = ...;

IClassManager.IConfig config = new IClassManager.CConfig(m_urlRegistry);

IClassManager.IContext ctxt = new CClassContext(getClass().getClassLoader());

IClassManager mgr = (IClassManager) fac.requestComponent (cId, config, ctxt);
```

Figure 4.24: Requesting a Component Object

In the example, a class manager component is to be created and configured with the URL of a class collection registry and contextualized with an object reference of the related class loader. As a result, a component may be seen as a special type of model resource that follows specified implementation guidelines like the feature interface approach. In the context of self-managing application composition, they can be used to facilitate the composition of complex objects.

4.3.2.5 Autonomic Application

The Java Loadable Modules approach can be used to implement *Autonomic Component Loading* and dynamic software composition by resolving application resource requests in a self-managing way, as shown in Figure 4.25.

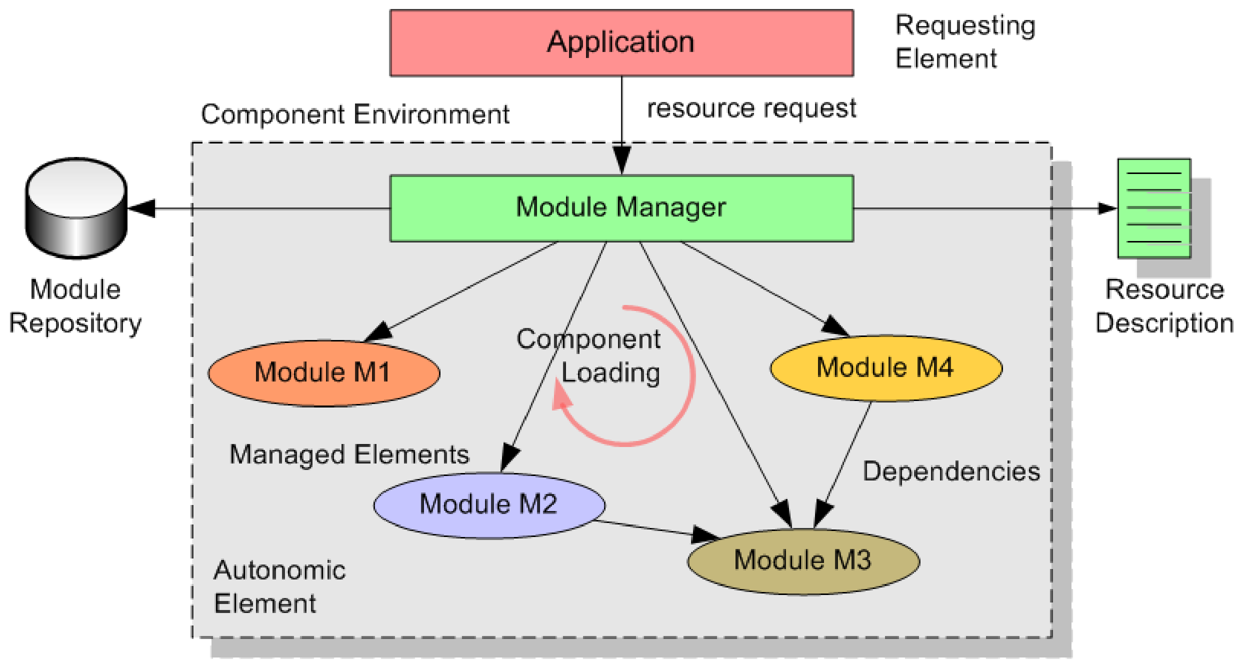


Figure 4.25: Autonomic Component Loading

A module represents an application composition unit that contains various resources and can be dynamically loaded during runtime. If an application requests a component resource, it passes the resource description to the module manager which queries for the resource across the loaded modules in the component environment. If no offering module is found, the module manager determines a suitable module using the module repository and the module descriptions stored there.

The autonomic operation is performed in a self-managing way by monitoring the requested resources in the component environment and controlling the module instances. From this point of view, the component environment represents the autonomic element, the application acts as the requesting element, the module manager is the autonomic manager and the modules are the managed elements. This allows adapting the selection and loading of modules according to the application configuration, platform capabilities and current composition scenario. For example, if an application is launched and required components have to be loaded, the module manager dynamically resolves and loads suitable modules, e.g. by following a policy to prefer and share already loaded modules *(self-configuration)*. Once an application composition has been performed, the module manager monitors the allocated resources of the related modules and unloads the instances that are no longer in use, e.g. if there are no resources used anymore. The next time a resource is requested, the module manager can recheck the composition set up and select a different module *(self-optimization)*. Furthermore, the module manager monitors the initialization of a module and the resources provision. If an error occurs, the module is unloaded and replaced with a different one matching the requirements. By using the object space approach [see Section 4.6.2], object references to unloaded resources are transparently restored after module replacement *(self-healing)*. Finally, the module manager protects modules to be untimely unloaded or manipulated by misbehaving applications, e.g. due to an attack. The same is valid for accessing private modules loaded by an application in a multi-tasking environment *(self-protection)*.

4.3.2.6 Related Work

The legacy Java programming model has not yet addressed the modularization of Java applications and the establishment of a distributed component management system. Based on the dynamic class loading approach, application composition is typically performed on a per-class basis. Over time, many approaches and implementations emerged to support software components and loadable modules. A possible taxonomy of software component models can be found in [214] and a related comparison of native component management capabilities in MS COM, Sun Java and MS .NET is discussed in [100]. The following review starts with a discussion about built-in composition features of a legacy JRE, than it gives an overview of approaches used by various composition frameworks and outlines the component management features of integrated application environments.

Legacy Java Runtime Environment. Since the original Java language specification does not address software composition and application modularity, there is ongoing work to extend the specification within the *Java Community Process* and the *Java Specification Request (JSR) 294* [49]. The basic idea is to embed modularity support in the Java language and to introduce so called *super packages*. As soon as this proposal will be implemented by Sun, as predicted for Java version 7, it will supersede proprietary component models such as proposed with the Java Loadable Module approach. In addition, existing solutions like *OSGI* and *Sun's JigSaw* will be probably adapted to conform to this component standard. Nevertheless, the support of flexible resource definitions and queries for resolving suitable components in a distributed operating scenario will not be an integral part of JSR 294. A related task is partially performed by the *JNDI* approach for acquiring and initializing pre-configured and shared resources, e.g. a database connection point. In this scenario, the providing module is either already loaded or can be resolved from a local module repository. Concerning native support of distributed component lookup and deployment, however, there is currently no plan to extend the legacy JRE into this direction.

Java Composition Frameworks. A common approach of Java composition frameworks is the introduction of an individual component model and the use of dynamic class loading to retrieve components on-the-fly. Well-known examples are *Sun EJB*, *OSGI*, *Spring* and *CORBA Components* [336, 188, 330, 28]. They allow composing applications from independently developed components for different purposes but still rely on the inherent deployment scheme of the underlying runtime environment. In other words, they do not introduce a deployment model offering the dynamic lookup and retrieval of distributed software components during runtime, e.g. by using a remote module repository and dynamic resolution of suitable modules. Thus, they are supposed to be primarily used with local component repositories and are not designed for hot component deployment in cross-platform application environments. There are various implementations that extend the basic capabilities with additional composition features. For example, an OSGI-based module loader has been implemented within the *Oscar* project [159] that uses a policy-driven class loader and separates the composition process from the class loading infrastructure. It focuses on the life-cycle management of components, allows dynamic updating of loaded components but it is still limited to the local class loading capabilities of OSGI. The Java module system *MJ* [69] introduces a module language to specify component dependencies and related tools to compile extra module statements added to the actual Java source code. It refactors existing Java source code to add particular composition features like import definitions and extension points. A step further towards component definition by using inline Java code annotations is performed in the *Spring framework* [330]. The component model supports annotation-driven dependency injection and various configuration files describe the resource definitions of a component. During runtime, the framework transparently evaluates the Java annotations and resource definitions to perform the application composition. The examples above outline the variety of application composition objectives and how related frameworks add extra composition features to Java applications. In general, the frameworks focus on the composition aspects and are based on managed module repositories on the same host. They are not supposed to dynamically deploy and retrieve modules and contained components from remote sites.

Java Application Environments. While a composition framework is typically built upon a legacy Java runtime environment and is usually deployed along with the application itself, this is not feasible for all kinds of applications. As an example, for concurrently managing web applications in a *server application environment* a so called application server is needed. It is started before the actual applications are loaded and then launches each application and service as part of the same process. Related applications are developed using a certain application model, e.g. *Java Servlets* or *Enterprise Java Beans (EJB)*, and can be solely executed in the target application environment provided by the application server; examples are *Jakarta Tomcat, Sun ONE, JBOSS, Apache Avalon* [141, 377, 179, 15]. While references to shared components provided by the application server are managed by using a common naming manager, e.g. *JNDI*, this is not valid for application objects. In detail, an application server typically separates application instances from each other and therefore a composition configuration is also related to a single application and does not take in account the life cycle of components of concurrently hosted application instances. Moreover, due to their orientation towards well-defined server-side scenarios with fixed system configurations, application server approaches have not been designed to be configured on-demand for the dynamic composition of new applications, e.g. due to user request. A well-known approach to organize software composition in a *remote application environment* is implemented by *Sun Java Web Start* [331]; it is part of the standard Java installation and eases the deployment of *Java Applications* in that it organizes downloaded Java archives in a locally managed cache. Each time an application is to be started, Java Web Start compares the cached version with the server version and downloads the application only if there is a newer version on the server. It inherently relies on Java archives (JAR) and though it extends the deployment scheme of remote applications, it does not come with a component model but it focuses on the hosting of legacy Java applications. Moreover, while it is a good starting point for distributed Internet application systems, it is not able to directly communicate with remote JAR repositories but by downloading and evaluating a *Java Native Launch Protocol (JNLP)* configuration file from a web server. And it also does not allow a customer to dynamically change the provided configuration of a JNLP application. An interesting approach towards software composition in a *distributed application environment* is provided by *Jtrix* that is not fixed to a single host but targets *code mobility* across multiple platforms [324]. It propagates so called *netlets* that represent a certain kind of Java service. They can be dynamically retrieved from a remote module repository and instantiated on the current platform as well as migrated and spread across different nodes. In this sense, Jtrix establishes a particular cross-node application environment with respect to nomadic services but therefore it only supports service-based deployment and composition, respectively. A related approach is *Cells* in that it introduces so called *deployable containers* [302]. It likewise encapsulates component code and object data that can be moved from one host to another. A high-level language *JCells* is used to define the component dependencies and interaction. While it has been mainly designed to support the distributed execution of Cell applications in the Internet, it is not suitable to run legacy Java applications without refactoring the source code according to the Cell programming model. Finally, *Cingal* is similar to JCells in that it also supports component deployment and the composition of distributed Java applications [83]. It shares the idea of OSGI to introduce so called bundles and various high-level configurations describing the application composition independent of the actual deployment scenario. Similar to JCells, it focuses on running distributed service components that communicate via network links.

In summary, there are plenty of dynamic software composition solutions available for Java. If Java Web Start is mainly seen as a software deployment solution without a distinct component model, none is yet supported by the legacy JRE. Thus, one has to choose a composition framework or application environment with composition features built-in. For implementing a cross-platform operating environment, however, additional features are needed that cannot be easily added to existing implementations. For example, the Java Loadable Module implementation heavily relies on the crosslet deployment features to dynamically resolve and download modules from remote module repositories. The managed organization of concurrent Java Class Spaces in a multi-application scenario allows the module manager to share and rebind components among application instances without putting them into the shared framework configuration. From this point of view, the Java Loadable Module approach represents a practical choice to add dynamic software composition features to the XDK without claiming to surpass OSGI or Spring. It lacks various features like advanced component lookup techniques to choose the best matching module [29] and it certainly introduces an increasing complexity to component dependency management in a large-scale distributed environment [201].

4.4 Shared Application Hosting

For better exploitation of hardware resources, regular computing systems usually run multiple applications and services that are managed in separate *hosting environments*, e.g. an operating system manages processes in distinct address spaces. Considering the concurrent hosting of Java applications, the memory and startup overhead for launching an individual Java Virtual Machine (JVM) has emerged various approaches towards shared application hosting, e.g. Java servlets and web services are organized in different servlet containers within the same JVM. When a Java application is started, the related container is created following the predefined runtime settings, however, without considering the current overall resource configuration of the JVM. This results in shared use of the JVM but separated resource management for every Java application, e.g. creating distinct XML parser engine instances. In Section 4.4.1, a new *Adaptive Resource Broker* is introduced that selects suitable resources by mapping *application roles* on *platform resources* [282]. The broker manages platform resources along with their descriptions and receives resource requests in terms of role definitions from applications. The roles are mapped to suitable resources by evaluating the current mapping policy. Besides platform resources, the sharing and shielding of application resources among application instances within the same JVM is another issue that has been only little addressed so far. In Section 4.4.2, so called *Java Task Spaces* are presented to organize private and public resources of application instances. In this context, the notions of *scene* and *stage* are introduced that reflect the collection of role definitions and resource configurations, respectively. In effect, applications are organized in scenes that are mapped on stages during application startup whereby applications on the same stage can mutually access each other's resources.

4.4.1 Adaptive Resource Broker

Typical application composition strategies are based on the direct mapping of resource queries on resource properties, e.g. querying resources by their name. In this section, a different approach towards self-managing resource binding is presented that maps application roles on platform resources by dynamically evaluating a composition policy. The realization of the approach using an adaptive resource broker is described and the application is illustrated.

4.4.1.1 Motivation

If an application is loaded into the hosting environment, it usually requests access to various platform resources, e.g. by getting a file handle for opening a configuration file or asking the window manager to create an application window. In turn, it may also create and offer resources, e.g. a callback handler for receiving system messages or a viewer component to be included in a document browser. The mutual access on application and platform resources is characterized by well-known interface declarations and methods that are used to bind the resources during runtime, e.g. a software component implements a given interface and the application asks a component manager to create a corresponding component instance. In this way, not only software components may be requested but any kind of managed program resource like established database connections or profile settings of a user.

The common aim of resource binding is to select and bind platform resources according to the application requirements. For Java applications, resource registration and binding is typically performed by using a runtime resource registry like approaches based on the *Java Naming Directory Interface (JNDI)* [343]. The resource provider registers the resource with a unique name and certain attributes while the resource requestor queries the registry for resources matching the query parameters. This works well for a single application environment if both the application implementation and the platform configuration are adjusted accordingly in advance, e.g. by the developer and the administrator during design time and setup phase, respectively. In a shared and alternating application environment, however, the static mapping of resource requests on resource registrations makes it difficult to find a common setup for matching the requirements of all installed applications at the same time.

4.4.1.2 Features

The goal of the *Adaptive Resource Broker* is to implement a new resource binding approach that dynamically resolves resource requests in a self-managing way by evaluating an adaptive binding policy. The major features are as follows.

Cross-Application Resource Framework. Besides the binding of resources provided by the current platform installation, concurrently hosted applications may share and request resources from each other. A cross-application binding framework allows registering application resources like platform resources. In both cases, the resource binding is transparent to the application that also enables the mutual sharing of dynamically resolved resources between unknown applications.

Dynamic Role Mapping. The role mapping approach is based on the dynamic evaluation of the application role descriptions and system resource definitions. A role manager receives the role description of the application and selects a matching resource by additionally considering the current runtime scenario, e.g. memory load and already instantiated resources. After all, a resource binding to an appropriate resource is established without particular application intervention.

Multi-Level Mapping Framework. An application may not seek for a specific resource instance but use any resource matching the required features, e.g. a newer version then originally requested. For this purpose, a multi-level binding framework introduces a role manager along with roles that are dynamically mapped onto a resource instance using a system-specific role description. The application does no longer requests resources but roles mapped on a matching resource later on.

Developer Task Decoupling. The introduction of roles enables the decoupling of developer and administrations tasks by using separated configuration files. While the component developer packages resources along with a related resource description in an assembly, the application developer specifies the resources required by the application using a role description. During runtime, the role description is read and a matching resource is retrieved on-the-fly.

Adaptive Binding Policy. The fixed mapping of roles on resources by using a hard-coded strategy hinders the system to adapt to new conditions in a flexible way, e.g. following different platform administrator preferences for distinct hosts. In the following approach, a casting manager uses an external policy definition to customize the role mapping in a self-managing way, e.g. dynamically selecting less-memory consuming components if many applications are concurrently run.

4.4.1.3 Approach

An application is composed of various resources that are provided by the application environment on request. Depending on the chosen programming paradigm, a resource may represent various units, such as a regular library or a component definition. Common to all types, an application does not need to know the actual resource implementation as long as the resource implements the expected interface, or in other words, it can play the requested role in the application environment. In a multi-tasking environment, various applications can request different roles that may be mapped on the same resource, e.g. a service instance implementing two interfaces. A corresponding approach is based on the introduction of an adaptive resource broker that mediates between resource queries issued by an application and resource implementations provided by the shared resource environment, as shown in Figure 4.26.

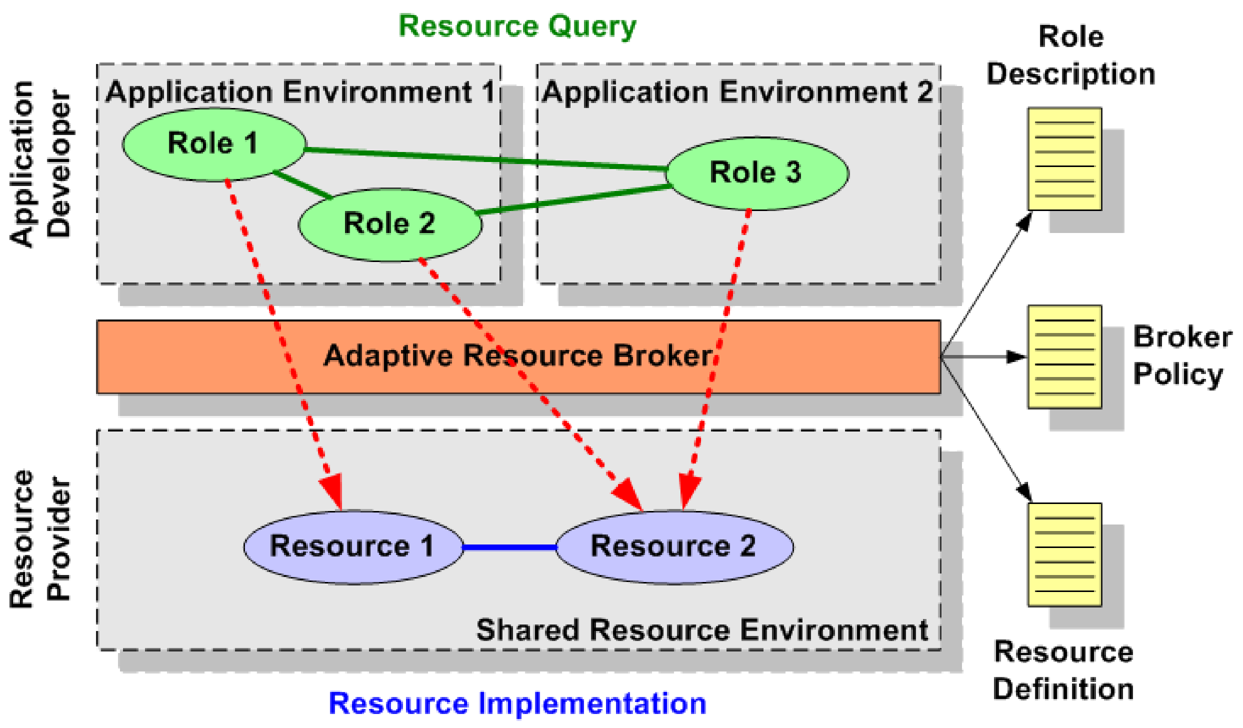


Figure 4.26: Adaptive Resource Broker

The resource provider implements various resources and provides a resource definition file describing the properties of the resource. In turn, the application developer designs an application by defining the roles a resource has to implement and providing related role descriptions. The adaptive resource broker receives the resource query and reads the role description. Then, loaded resources are checked for matching the given role and if none has been found, the resource definitions are evaluated. A broker policy can be used to adjust the role-resource mapping by the system administrator, e.g. excluding certain resources from being shared in multi-tasking environment or refining the resource query with additional parameters.

4.4.1.4 Realization

The realization of the approach is implemented around a multi-level resource mapping framework which is customized by various configuration files, as shown in Figure 4.27.

The application issues a resource request by using a unique role id that is passed to the role manager and resolved using an external configuration file `roles.xml`. After having retrieved the role description, the role manager passes it down to the casting manager that is responsible to find a suitable resource. To this end, it queries the resource manager in the next layer beneath for a matching resource. The concrete strategy and policy to determine a matching resource is not fixed but adjustable using an external configuration `policy.xml`.

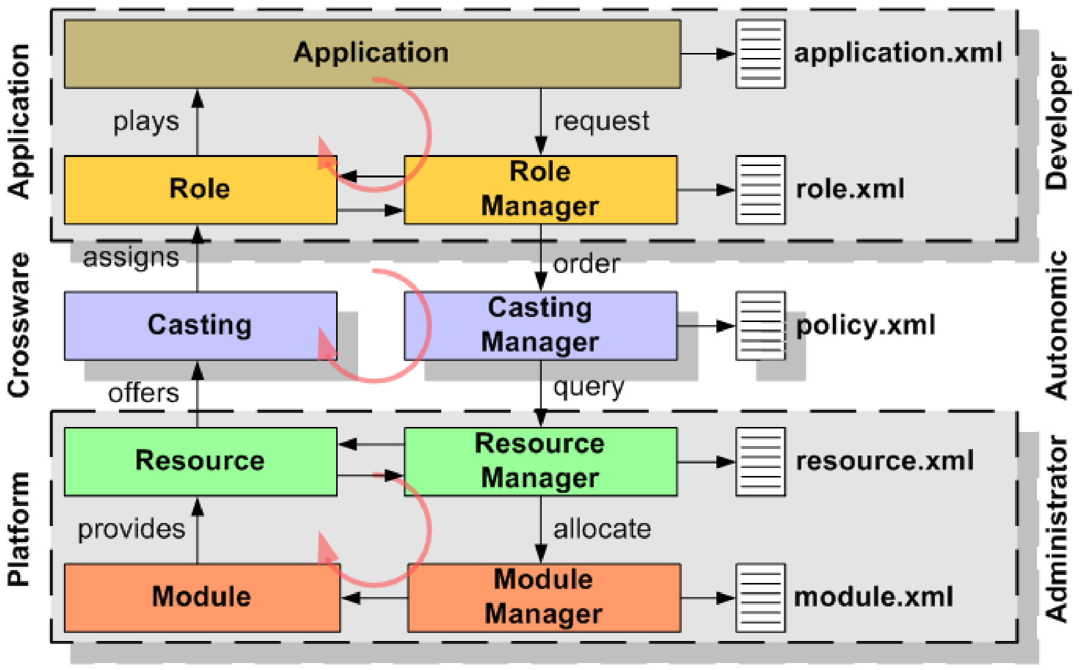


Figure 4.27: Multi-Level Resource Mapping Framework

If a suitable resource has been found, the casting manager is requesting the resource from the resource manager using the unique resource id found in the `resource.xml`. In the next step, the module manager is asked to look up the related module in `module.xml` providing this resource, and if it has been not loaded before, it will be dynamically requested. The module provides access to the required resource and returns an object reference up to the resource manager. The casting manager takes the object reference and encapsulates it into a casting object that is further passed as a role to the application. In this context, the casting object acts as a controlling proxy for the self-managed application composition and mapping role-based interfaces to resource-based implementations.

Overall, the application and the role manager, as well as the resource manager and the module manager, form independent control loops. In the middle, the casting manager is like an agent at a theatre keen to map roles to suitable resources. It does not deal with the concrete composition request or the deployment scenario but attempts to find a platform resource according to its strategy and a specific application request.

Configurable Resource Selection. The module approach enables to query and open modules by their properties [see Section 4.3.2]. However, the application is typically not interested in opening a certain module but in requesting a particular resource. On top of the module manager shown in Figure 4.27, a resource manager is added to resolve the module that contains the requested resource using an external configuration file `resource.xml`. The resource manager does not know how to load the modules. It only resolves the module id and passes it to the module manager. If the module has been already loaded, the module manager simply returns an object reference to the module. An excerpt of a property based resource configuration is shown in Figure 4.28.

```
<resource id="{57F2B411-1171-4fe2}">
  <module id="{7A861A5D-42A1-425d}" />
  <property name="vendor" value="crossware"/>
</resource>

<resource id="{57F2B411-1171-4fe2}">
  <module id="{D0FCC9C7-E078-4278}" />
  <property name="vendor" value="uni-siegen"/>
</resource>
```

Figure 4.28: Property-Based Resource Description

The resource with the id `{57F2B411-1171-4fe2}` is offered by two modules. If the application passes only the resource id, the resource manager will select the first module containing the resource and which has been already loaded. Another option would be to pass additional parameters to select a resource by its properties, e.g. matching a certain vendor. As a result, the application does no longer need to know which module provides the resource but can simply pass the resource id to the resource manager and will receive the resource, as shown in Figure 4.29.

```
IResourceManager rmgr = ...

CResourceId resId = new CResourceId("{57F2B411-1171-4fe2}");

Class c = (Class) rmgr.requestResource(resId);
```

Figure 4.29: Requesting a Class as Resource

In this example, the application passes the resource id, and the resource manager is resolving a module in the background and returns an object reference to the requested resource, e.g. a class object.

Adaptive Resource Selection. Next, the notion of roles and the self-managed mapping to resources with a casting manager are introduced. Instead of knowing the resource that provides a certain function, the application refers to roles that are not linked to any resource but are supposed to be dynamically mapped on an implementing resource, e.g. by matching properties such as type and version, as shown in Figure 4.30.

```
<role id="{8711B3B0-56BB-4f55}">

    <property name="type" value="classmanager"/>
    <property name="version" value="1.0.2"/>

</role>
```

Figure 4.30: Property-Based Role Description

The application requests a role from the role manager that queries the role configuration for the description of the role, as shown in Figure 4.31.

```
IRoleManager rmgr = ...

CRoleId roleId = new CRoleId ("{8711B3B0-56BB-4f55}");

Class c = (Class) rmgr.requestRole(roleId);
```

Figure 4.31: Requesting a Class as Role

The actual mapping of a role to a suitable resource is performed by the casting manager. It receives the resolved role description and returns a matching role, as shown in Figure 4.32.

```
public IRole cast(IRoleDescription roleDesc) throws XNoMatchingResource
{

  Collection props = role.getProperties();
  IResourceDescription desc = new CResourceDescription(props);

  Collection coll = rmgr.requestResource(desc);
  if (coll.size() == 0)
  {
    throw new XNoMatchingResource(roleDesc);
  }

  return (IRole) coll.iterator().next();
}
```

Figure 4.32: Simple Property-Based Casting

In the example, the casting manager retrieves the properties of the role description and encapsulates them into a resource description. The exemplified casting strategy is quite simple and just compares the properties one-by-one. In a real-world scenario, the approach uses custom casting plugins, e.g. evaluating additional parameters such as required classes or considering environment settings of the host platform.

4.4.1.5 Autonomic Application

The Adaptive Resource Broker approach can be used to implement *Autonomic Resource Binding* in a cross-platform operating environment and with concurrently hosted Java applications, as shown in Figure 4.33.

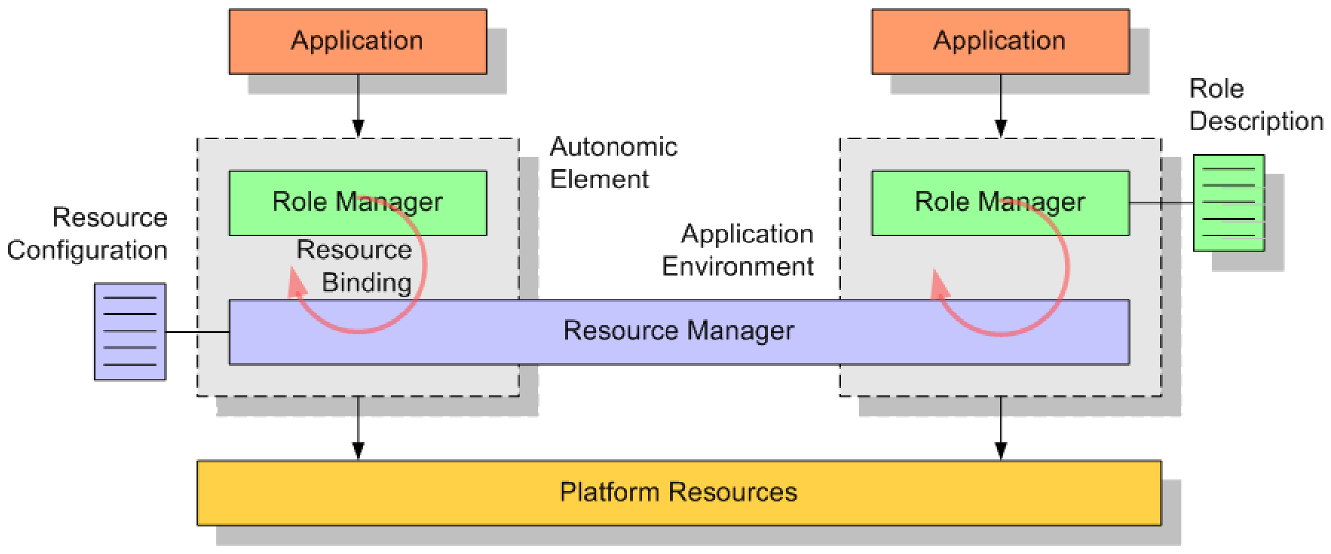


Figure 4.33: Autonomic Resource Binding

A platform resource is a non-moveable entity, e.g. a database instance or a GUI framework, which may be bound by the application during runtime. To request a resource binding, the application passes a role description to the role manager. In the case, there is a suitable resource already bound, e.g. a database connection has been already opened, the role manager will return the existing binding. Otherwise, the resource manager is directed to determine a matching resource.

The role manager performs the autonomic operation by monitoring and controlling the resource bindings. From this point of view, the application is the requesting element, the application environment represents the autonomic element, the role manager acts as the autonomic manager and the resource manager is the managed element. Concurrently hosted applications do no longer requests resources directly but use a common resource manager to synchronize resource requests. The application-specific role manager tracks existing resource bindings and is able to adapt role requests according to the current binding scenario. For example, if an on-demand application is deployed and launched on a new computing system, suitable platform resources are determined by evaluating the role descriptions and platform configuration, e.g. discovering a graphical user interface to present an advanced application window *(self-configuration)*. During runtime, the role manager checks the resource bindings and may modify an existing binding according to a given plan, e.g. increasing the data cache size in case of increased cache miss rates *(self-optimization)*. Similar to this scenario, the role manager can also determine failing resources, e.g. by periodically polling the health state of a resource or introducing an interceptor as sensor [see Section 4.6.1]. For example, a database operation may fail due to a shutdown of the database engine and a new database connection is reopened without application intervention *(self-healing)*. Further, the role manager can check the authenticity of a resource and protect the application against using an exchanged resource item, e.g. in case of passing a private key to an encryption resource *(self-protection)*.

4.4.1.6 Related Work

Driven by the large-scale deployment of applications in heterogeneous environments the need for adaptive applications came up and various approaches were proposed to cope with conflicting requirements in distinct application domains [40, 233]. Besides the dynamic adjustment of the program execution, this also includes the runtime reconfiguration of the application structure. A related survey of distinct *adaptation spaces* can be found in [7]. While there are application-level proposals that require the extra preparation of the application code to adapt its behaviour, there are system-level approaches that introduce runtime adaptation as part of the software architecture [37], e.g. a component framework or a reflective middleware. The following overview focuses on compositional adaptation and reflective adaptation approaches that follow the *separation of concerns* and enable the adding of related self-management features on the system level without application modification [30].

Compositional Adaptation. The component-based development paradigm enables the dynamic composition of software applications. This can be used to select and deploy software component according to the application scenario during runtime. Common composition approaches, such as *Enterprise Java Beans (EJB)* [242] or *CORBA Components* [232], perform the actual composition by evaluating the application configuration, the runtime environment and particular component deployment descriptors. However, these approaches are typically designed to support separated application execution and there is no way to adjust the binding of selected component resources, e.g. to share selected component resources with concurrently loaded. This results in an all-or-none resource binding configuration. In addition, the rebinding of system resources is not supported, e.g. after an application instance has been migrated from one platform to another. To address this issue, advanced binding approaches have been proposed like *Colomba* [30]. Originally designed to support dynamic binding in mobile applications, its binder manager and policy managed can be used to separate application logic from resource binding management. This is similar to the *Adaptive Resource Broker* approach. It also exploits resource metadata and uses an advanced policy specification to control the runtime adaptation in dynamic environments and to hide low-level binding mechanisms from application developers and system administrators. While Colomba represents a powerful resource binding solution to migrating mobile applications, it focuses on service resources and misses particular support for resolving component resources and the retrieval of suitable software components. Moreover, Colomba is build on top of *SOMA* [321], a mobile agent platform, and cannot be easily integrated into another middleware implementation, e.g. into the XDK. Another direction of compositional adaptation is the dynamic and transparent modification of component code while it is loaded, such as provided by *JOIE* [63] and *JMangler* [200]. They allow to respond to changed interface specifications and to adjust the corresponding component interaction. Although these approaches support the transformation and weaving of random components, they do not address the autonomic selection and binding of compatible component resources with respect to the platform configuration and computing scenario. In addition, they add another level of complexity to the overall system may easily produce malicious component implementations.

Reflective Adaptation. In contrast to compositional adaptation which typically evaluates extra metadata configurations, reflective adaptation relies on the introspection of component implementations and the introduction of a *meta space* [40, 70, 71]. They basic idea is to open a programming interface to the component implementation that can be used by a development tool or runtime middleware to adjust the composition process. There are related approaches, such as *ARCAD*, *AspectJ, OpenJava, R-Java* or *TRAP/J* [80, 233, 310], which introduce custom compilation models based on meta objects. They support particular composition features during design time but do not support the adaptive configuration of application composition during runtime, e.g. due to changed runtime conditions. Other approaches, such as *OpenORB, OpenCOM, Iguana/J* and *Prose* [38, 62, 233], use open implementation and computational reflection to inspect the components and to adjust the binding process during runtime. They offer additional features to the middleware like the option to monitor the internal resource condition and to reinitialize the resource according to changed runtime constraints, e.g. available memory and CPU utilization in a shared application environment. While this facilitates the low-level adaptation, reflective approaches require particular component implementation and typically suffer from offering too much flexibility [37]. Moreover, the reflective code is part of the component implementation and therefore it can only be customized if the corresponding source code is available.

In summary, related work has shown a particular need for reconfiguration and recomposition of software applications. While application-level approaches require additional implementation effort and are limited to a single application instance, system-level approaches represent a good choice to implement an adaptive resource broker approach as part of a self-managing middleware. The comparison of selected compositional and reflective adaptation solutions has revealed their differences concerning easy system integration and adaptation flexibility. Reflective adaptation solutions offer a greater degree of *behavioural customization* but they typically experience performance problems in large and dynamic application systems [37]. Compositional adaptation solutions focus on the *structural customization* and are less intrusive concerning the overall implementation than reflective adaptation approaches. In particular, policy-based solutions like *Colomba* [30] add configurable runtime adaptation capabilities while separating the resource binding management from the application logic. From this point of view, various Java adaptation solutions are available that may be integrated into the XDK to manage resource binding in a self-managing way. A common drawback of the related solutions, however, is their original design to focus on single application adaptation and the lack of adjusting the configurations of concurrently loaded applications. As a result, the Adaptive Resource Broker represents a compositional adaptation solution that offers a unique multi-application configuration without imposing the complexity of reflective adaptation solutions.

4.4.2 Java Task Spaces

To support multi-tasking program execution in regular Java runtime environments, an essential requirement is the configurable management of system and program resources. In this section, *Java Task Spaces* are introduced to manage private task resources and to handle provisions and requests of shared resources. For this purpose, a self-managing resource assignment scheme is implemented by mapping application-specific resource role definitions on platform resource implementations.

4.4.2.1 Motivation

If an application is to be started, the host system creates an application environment in which the application code is loaded and run. In a single application environment, only one application is executed at the same time. Application composition, resource handling and task management are fairly simple like in the original Java Runtime Environment (JRE) approach. Things change if various applications are concurrently hosted like in a shared application environment, e.g. MS Windows or Linux. The runtime system typically creates separated address spaces for hosting each application instance, and managing each application and its resources separately. The access on common system resources is controlled by the runtime system and mapped into the address space of the application. Since the mutual access of application resources is also typically prevented, there are various approaches, like *Shared Memory* and *Pipes*, to support the collaboration of distinct applications.

For Java application environments, shared application hosting is originally not supported though there are various approaches out there that mimic the behaviour as good as possible. Since the original JRE does not support separated address spaces within the same Java Virtual Machine (JVM), popular approaches like servlet container or application servers create multiple application class loaders that enable the separation and shielding of application classes in a multi-tenant operating scenario, e.g. hosting various customer services. However, this is limited to managing *class resources* but does not actually introduce address spaces to manage access on random *task resources*. Moreover, there is no support for multi-task handling in a JVM, e.g. for associating thread groups and threads to an application instance, also known as task. Due to this, the JVM lacks a task manager and the ability to fork sub tasks that share resources with the parent task, e.g. for launching a new terminal session.

4.4.2.2 Features

The goal of the *Java Task Space* approach is to enable the multi-tasking operation of a regular JVM by managing concurrently executed Java tasks and their resources in a multi-tenant fashion. The major features are as follows.

Task Resource Management. To handle multiple tasks in the same JVM, related application code, task resources and objects have to be managed and associated with a task, e.g. code assemblies, execution threads and task objects. The Java Task Space approach offers the separate and shared use of task resources, e.g. between concurrently hosted applications or a task and its forked child tasks.

Java Task Manager. To administer task units in a multi-tasking JVM, a task manager keeps track of all running applications and enables users to start, stop, suspend and resume selected Java tasks. Starting with the first task, the task manager organizes subsequently launched child tasks in a hierarchical manner whereby a parent task and its child tasks form a common task space; hence grouping related tasks and their resources.

Reentrant Code Execution. When a new task is to be started, the task manager checks if there is a running instance of the corresponding application and offers users to reuse the existing task space. For particularly designed applications, code components and static resources may be shared between tasks while each task unit can create private task resources, e.g. a text editor can reuse the spell checker but works on a different text document.

Multi-Session Support. Besides multi-tasking support, the XDK offers multi-session handling and to it, task spaces are used to associate running tasks and their resources to a login session. In particular, task spaces enables tasks launched by the same user to register and query session-related resources, e.g. for sharing a graphical desktop interface. Whenever a session ends, the task manager stops all tasks in the related task space and releases its resources automatically.

Custom Task Collaboration. Similar to *Inter-Process Communication (IPC)* in conventional operating systems, the XDK offers tasks belonging to different task spaces and sessions to communicate and exchange data. Java tasks may query the task manager for a certain task and open an object queue to it. By using particular object serialization features and dynamic class resolution as presented in Section 4.6, objects may be passed between tasks without exposing private object references.

4.4.2.3 Approach

To run a Java application, a JVM is started and the program execution starts by executing the method `main` of the application class passed on the command-line. In this single-tasking scenario, the JVM acts as task manager and session manager at the same time. There is only one task environment and a task is simply comprised of all running threads, initialized resources and loaded application code. To transform the JVM into a multi-tasking and multi-session environment, the following approach introduces a task manager that creates task spaces for managing distinct task objects, a resource manager for registering shared and private application resources and a session manager for handling multiple user sessions, as shown in Figure 4.34.

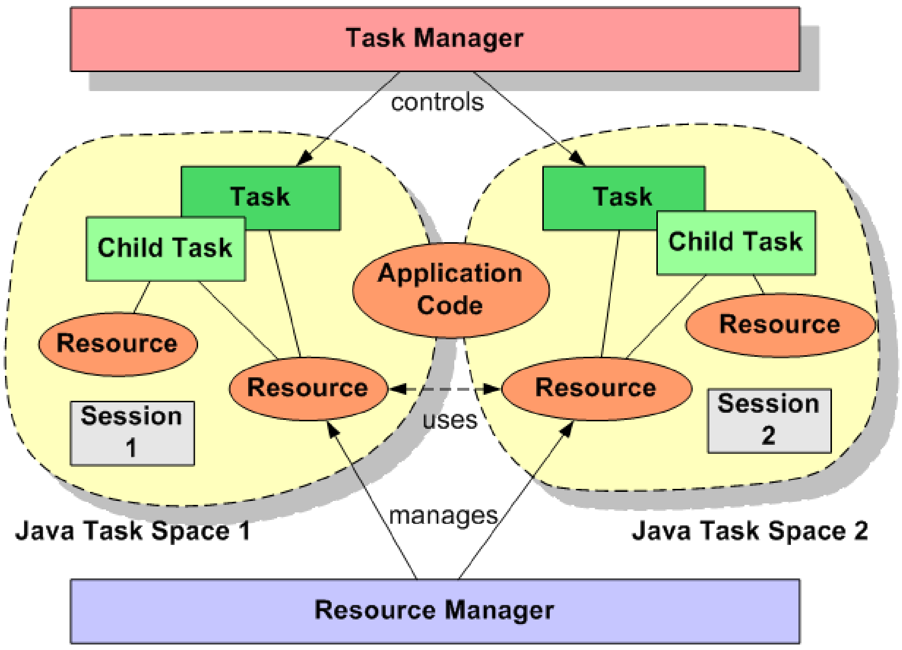


Figure 4.34: Task Management using Java Task Spaces

When the JVM is launched, first the resource manager, the session manager and the task manager are initialized to control multi-session and multi-tasking operation. If a user session is created and the first application is started, a new task and task space are created, and associated with the user session. Additional tasks may be launched reusing and re-executing the same application code, or run in a separately customized application environment [see Section 4.5.1]. Either way, every new task is a child task of the launching task and associated with the same session. Thus, a task hierarchy is built and controlled by the task manager, e.g. when stopping a parent task, all child tasks are stopped as well. The same is valid for terminating a user session using the session manager. For task collaboration, a task can register task resources with the task space that can be queried by tasks in the same task space or concurrently hosted tasks in different task spaces, e.g. a service task may provide a programming interface for controlling the installed service.

4.4.2.4 Realization

The realization of Java Task Spaces is mainly driven by separating the concerns of *task management* and *resource management* in a multi-session resource environment, as shown in Figure 4.35.

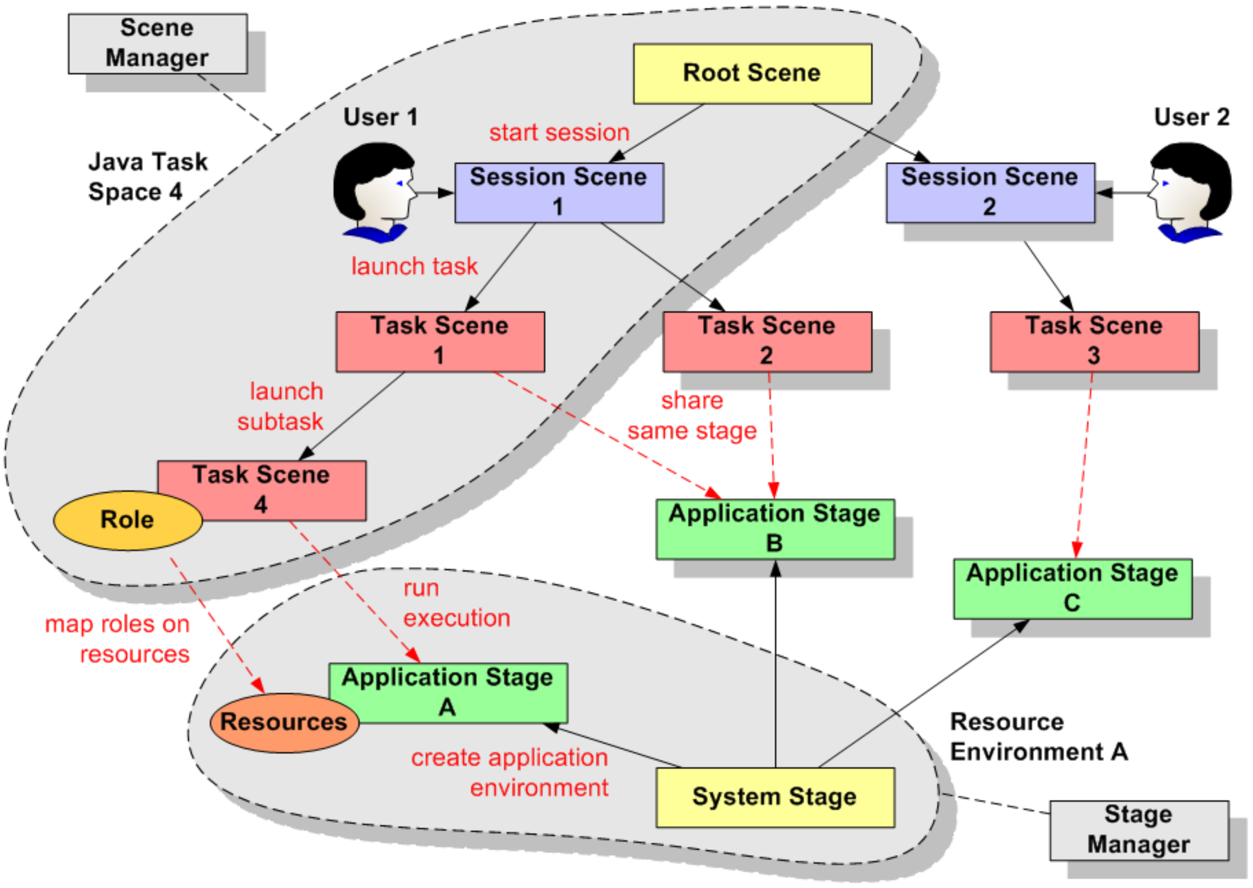


Figure 4.35: Task and Resource Management in a Multi-Session Runtime Environment

Before a Java application is actually launched and the present task is processed, the application launcher configures an appropriate application environment to host the required application resources like Java classes, modules and components. In this context, the Java Task Space realization introduces so called *stages* to manage system and application resources, e.g. module loader and component factories. The task instance running the application and related task objects are managed by so called *scenes*. This implementation enables the processing of one or more tasks using the same application resources, hence playing the same application in various task scenes on a common application stage.

As tasks launch sub tasks and create a task tree of parent and child tasks, related scenes are organized in a hierarchical manner whereby all scenes on the path to the *root scene* via the *session scene* create a common task space, as shown in Figure 4.35. The root scene is the scene created when the first application is launched by the XDK, e.g. the Internet Application Workbench [see Section 5.3]. The session scene 1 is related to the user session 1, and task scene 1 and task scene 4 are associated with the task 1 and its sub task 4. As a result, the task 4 can view and access all task objects of its parent tasks and share session objects of the related session with task 2 managed by task scene 2. This results in the reuse of application resources while still separating the management of task and session objects.

Similar to scenes, stages may create child stages that allow sharing common application resources, e.g. peer registries and crosslet repositories, and shielding private application resources, e.g. code assemblies and component factories. A particular stage is the *system stage* that is created during the initialization phase of the XDK and is configured with a class space configuration containing all core classes of the JDK and XDK. In addition, the system stage is used to register system-wide platform resources like the session manager or the task manager. All other stages are usually created as a child stage of the system stage when a new application environment has to be set up, as shown in Figure 4.35. Finally, the application resources of a stage and its parent stages form a common resource environment that allows a task to transparently access private application resources registered with the associated stage and reuse shared application resources provided by the parent stages.

Stage and Scene Contexts. If an application instance is started, the application launcher passes various context references for accessing the scene and stage objects. The references are passed by using an optionally implemented feature interface `IContextualize`, as shown in Figure 4.36.

```
public class CExampleApplication implements IContextualize
{
  private IStageContext m_stageContext;
  private ISceneContext m_sceneContext;

  public void contextualize(IContext ctxt)
  {
    if (ctxt instanceof IStageContext)
    {
      m_stageContext = (IStageContext) ctxt;
    }
    else if (ctxt instanceof ISceneContext)
    {
      m_sceneContext = (ISceneContext) ctxt;
    }
  }
}
```

Figure 4.36: Stage and Scene Contexts

Application Environment Resources. The context `m_stageContext` allows all application instances running on the same stage to register and share application-related resources as well as system-related resources. In Figure 4.37, an example for retrieving the global session manager and a list of application resources is shown.

```
// application environment resources
IStage stage = m_stageContext.getStage();
IResourceManager rm = stage.getResourceManager();
ISessionManager sm = rm.requestResource(ISessionManager.RESOURCE_ID);
Collection<IResource> applicationResources = rm.getResources();
...
```

Figure 4.37: Application Environment Resources

Session Resources. Every task is run in the context of a user session and the associated application instance may retrieve session-related resource objects that are shared by all application instances of the same user session. To access session-related resources, the session object is requested from the session manager, as shown in Figure 4.38.

```
// session resources
IScene scene = m_sceneContext.getScene();
CSessionId sessionId = scene.getSessionId();
ISession session = sm.getSession(sessionId);
Collection<IResource> sessionResources = session.getResources();
...
```

Figure 4.38: Session Resources

Task Space Roles. While an application may directly request resources, another option is to retrieve the role manager of the current scene and access application resources by using role definitions. In Figure 4.39, the example shows how to get a list of all registered roles in the task space and how to request a specific resource mapped on the given role.

```
// task space roles
IScene scene = m_sceneContext.getScene();
IRoleManager rm = scene.getRoleManager();
Collection<IRole> taskRoles = rm.getRoles();
IDesktopManager dm = rm.requestRole(IDesktopManager.ROLE_ID);
...
```

Figure 4.39: Task Space Roles

The hierarchical organization of stages and scenes is hidden from the application and resource and role manager are transparently traversing parent instances for determining requested resources.

4.4.2.5 Autonomic Application

The Java Task Spaces approach is used to implement *Autonomic Resource Sharing* between distinct application instances that are concurrently hosted in a multi-tasking runtime environment, as shown in Figure 4.40.

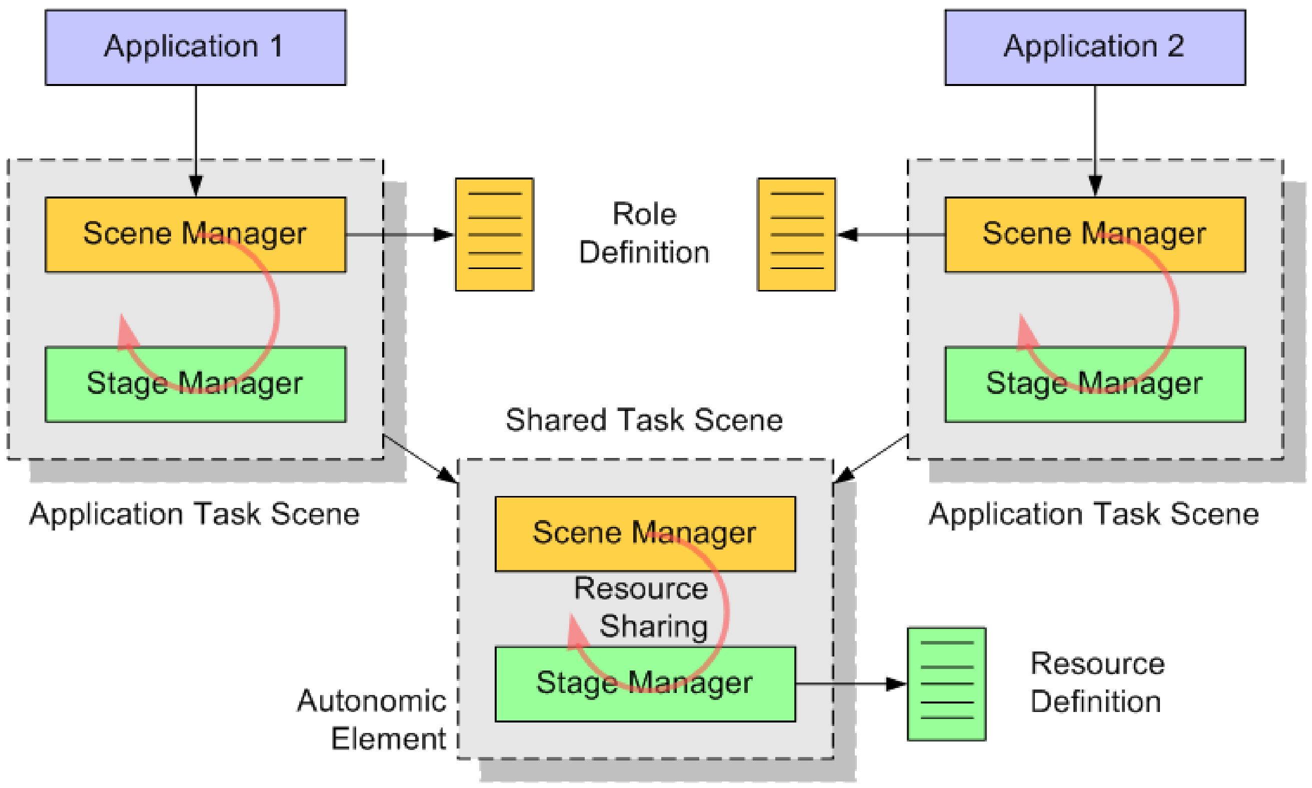


Figure 4.40: Autonomic Resource Sharing

In contrast to platform resources that are configured during system setup, task space resources are dynamically created and destroyed as applications are launched and terminated, e.g. a task-specific network protocol handler. By organizing the task spaces using hierarchically arranged scenes and stages, an application can decide which resources should be shared or kept private. For example, an application launching a desktop GUI could provide shared access to the desktop window but separate the application windows from each other.

The autonomic operation is performed by tracking and controlling the resource provisions and requests within a task space. From this point of view, the task spaces are autonomic elements, the scene manager represents the autonomic manager and the stage manager with the task resources are the managed elements. In particular, various scene managers of the task space are combined to build a multi-level autonomic resource sharing systems. If a task requests a certain resource, the scene manager of the origin task scene checks the current scene for a matching resource, and if none is found, it follows the chain up to the root task scene. In this context, every scene manager acts as an autonomic manager of the parent scene and monitors resource provisions of the related stage. For example, if a desktop GUI framework is launched and subsequently started applications should be displayed on the same desktop window, the framework registers the desktop window as a resource in its stage. The desktop applications are started in sub scenes and can access the shared desktop window by querying the parent scenes *(self-configuration)*. In this scenario, concurrent hosted applications usually create separated stages to which their code base is associated, e.g. related class spaces are configured with the application classes. A particular scenario is the start of several instances of the same application. The scene manager of the desktop framework can create distinct sub scenes on top of the same stage and share the same code base for minimizing the memory foot print and startup time *(self-optimization)*. In case of an application failure, e.g. due to a missing class, a separate stage can be created and configured with a different code base to run the application *(self-healing)*. The scene and stage managers can also be used to implement adaptive resource access control, e.g. granting access on resources of the parent stage to selected user sessions only *(self-protection)*.

4.4.2.6 Related Work

The goal of the Java Task Space approach is the extension of the legacy JVM to support multi-tasking operation with particular concerns to task-specific resource handling. This is related to Java application isolation in common and the organization of concurrent access to shared and shielded program resources. Although the actual idea has been discussed for several years [165, 338], there are only few public available implementations [348, 93] and in fact no widely adopted solution is known in practice. A good introduction to various forms of Java application isolation is provided in [74] along with a possible categorization of related work that is used in the following review.

Class Loader Based Approaches. The first category encompasses solutions based on a legacy JVM and a *class loader per application* configuration. Typically, an application framework is launched first that loads every application into the same JVM. The application isolation is realized by associating a separated class loader to every application and the assumption that no object reference is directly passed from one application instance to another. Well-known solutions are *Apache Tomcat* or *JBoss* [16] that typically host trusted web services and applications in the same JVM. While they support shared resource provision to the servlets, e.g. by registering and activating resources via *JNDI* [343], they lack support for associating certain resources to a running servlet instance. For example, Apache Tomcat may launch and unload multiple servlets that request distinct resources during their life cycle. The loaded Java classes are related to a servlet-specific class loader and thus can be separately managed, e.g. by using a servlet class loading policy [159], however, this is not valid for servlet-specific resources. As a result, acquired resources like an open file stream will still remain locked in case of an unexpected servlet shutdown. This issue is addressed in *J-Kernel* by introducing *capabilities* as handles to resources in another application domain and associated with a different class loader [165] A related solution tackles this problem by using so called *service gateways* that control resource management in a multi-service environment [307]. Based on the OSGI technology, a core service gateway is installed that can be compared to the task stage in the task space approach and multiple virtual service gateways that relate to task scenes. A similar solution is *Echidna* [93] that tracks acquired resources that are automatically released as soon as an application instance terminates. The common idea is to separate the resource provision code from the resource allocation code by introducing a specific programming model and by managing distinct class loaders for every application instance.

Process Based Approaches. In the second category, multiple Java applications are launched in separate JVM processes while allowing them to share a common memory region to reuse application classes and data [76, 87]. Since the introduction of *class data sharing* in Java 5.0, this approach is generally used by the native JVM when launching more than one Java application [337]. Besides enabling strong application isolation, this feature also reduces overall memory footprint and startup time of multiple application instances, e.g. in a desktop computing or on-demand service provisioning scenario. Moreover, the execution of one application does not affect concurrently running applications as in a shared JVM, e.g. no general blocking if a single application is locked in an infinite loop. In contrast to class loader based solutions, most of the regular Java applications work with this approach without any code modification or recompilation. While process-based approaches greatly exploit the OS-specific process isolation to run multiple Java applications, it inherently lacks native support to manage custom program resources across distinct application instances and process boundaries, e.g. sharing session data objects in a multi-user scenario. In addition, the launching of a separate JVM process per task represents a heavy-weight operation in comparison to the class loader-based approaches, especially with many short-lived applications launched in an on-demand computing scenario.

Java Runtime Modifications. The third category is related to approaches based on a custom JVM that is modified to offer particular application isolation features. The basic idea is to exploit and extend internal data structures to support transparent multi-tasking in a shared JVM. There is an ongoing discussion in the *Java Community Process* about extending the legacy JVM with an application isolation API [338] but there is yet no public implementation available. Another development line of Sun towards a *Multi-Tasking Virtual Machine (MVM)* [77] has not been adopted by the official JVM, mostly due to the fact that if a MVM process crashes. In this case, it will take down all hosted Java applications which is not feasible for a real-world business scenario. The same is valid for a proposed resource management interface for the MVM that is able to model application-specific resources and management policies [78]. From the agent and mobile computing research fields, various mobile agent systems have been proposed to support the mobility of executing programs. For example, *NOMADS* [348] introduces a custom Java virtual machine in which each agent executes in a separate virtual machine thread while all still run in the same process. In a multi-tasking scenario, this enables the sharing of common resources and the shielding of application-specific elements like code components and session data objects, respectively. However, agent and mobile application systems are designed to enable the transparent and isolated application execution. There is typically no support to manage task-specific resource requests among concurrently executed application instances, e.g. sharing a common desktop control element in a self-managing way as possible with the Java Task Space implementation.

In summary, true multi-tasking support is a long awaited feature for the Java runtime environment. Various approaches have been proposed to mimic the concurrent execution of Java applications and the management of private and shared application resources in a common runtime environment. A basic idea to enable multi-tasking is the isolation of Java applications instances and related resources. In a legacy JVM, this is not feasible since there is yet no programmable option to define a task and to separate task-related resources as proposed in the Java Task Space approach. Class loader based solutions at least try to achieve this for application class resources and some offer advanced resource management features like life-cycle management in *Echidna*. Nevertheless, there is still no task management available, e.g. for launching a subtask within the same JVM, and the scoped management of resources is also missing, e.g. sharing the resources of a user session among all applications of the same user while separating private application data objects. While using a legacy JVM to host multiple applications, a basic issue is its inherent characteristic to run everything in a single process and therefore without exploiting the process isolation features of the underlying host operating system. For critical applications, this may cause problems concerning security and stability requirements. From this point of view, true application isolation can be only achieved by launching a separate JVM process per task. While related solutions still lack advanced task and resource management features, they result in heavy-weight multi-tasking systems. Custom

JVM implementations like the *Multi-Tasking Virtual Machine (MVM)* can offer a reasonable compromise between strong application isolation and light-weight task handling, once they are widely adopted. As a result, advanced task and resource management in Java is still subject of various research and engineering efforts. The Java Task Space implementation offers autonomic resource sharing features as implemented with the hierarchically organized scene and stage managers. It is based on a legacy JVM and can be used to implement a light-weight multi-tasking system for non-critical use cases.

4.5 Pervasive Environment Customization

Before a software application can be run, a suitable *runtime environment* in which the application can be hosted has to be prepared by the runtime installer [see Section 2.3.3], e.g. by installing a servlet container for running a Java servlet. In a personal computing scenario with only one computing device involved, this task is easy to accomplish by the end-user, e.g. by running an installation CD. However, in a distributed computing environment, the user has to repeat the installation procedure on every computing device and to consider different platform configurations not known in advance; making it quite tedious and complex as well. In Section 4.5.1, an *application execution engine* is introduced that evaluates the application configuration and performs the preparation of the required runtime environment in a self-managing way while considering the present platform configuration and runtime policy, e.g. downloading and starting an OSGI container [284]. In this context, a particular issue is the customization of the deployed runtime environment according to the application, user and system profiles, e.g. choosing a specific proxy server for connecting remote web services. In general, the customization definitions are stored in a local profile repository on the computing device from where they are retrieved each time the user logs in and launches the application. While there are approaches to synchronize the profile across various computing systems in a uniform and well-known network environment, e.g. by synchronizing it with a master copy on a MS Windows domain controller, it does typically not apply to Internet computing systems with changing system configurations. In Section 4.5.2, *roaming user profiles* are introduced that are not bound to a central profile server and are evaluated by a profile recommender system with respect to the current requirements, e.g. a not yet existing proxy server configuration is derived from another user profile [283, 297, 362].

4.5.1 Application Execution Engine

The installation and configuration of a runtime environment for launching an application are extra tasks that are usually not performed by the regular user but by the system administrator. In the following sections, an *application execution engine* is presented that evaluates the application and platform configuration, selects suitable crosslets for launching the application and prepares the required runtime environment in a self-managing way.

4.5.1.1 Motivation

To execute a software application, a suitable runtime environment is needed that is able to load and run the application code, e.g. a MS Windows operating system. Since the related installation procedure is a non-trivial task, it is typically performed by the system administrator in a manual way, e.g. by selecting a matching release for the current computing system and following specific setup instructions. While this is feasible for basic operating system installations, however, there are various approaches that enable the installation of *lightweight runtime environments* like web browser plugins for launching thin clients. Apart from notifying the user about the installation, the setup is performed mostly without further user intervention, e.g. by installing a Sun Java plugin for running Java applications. For this purpose, the plugin vendor provides a download site with prepared runtime releases for well-known platform configurations and operating systems.

By applying the lightweight runtime installation approach on cross-platform operating scenarios, applications on heterogeneous computing systems can be launched as they are requested along with the needed runtime environment. For this purpose, an execution engine is needed that evaluates the application requirements and current platform configuration to determine a suitable runtime environment and, if needed, to download and install it automatically. Though there are working solutions like *Sun's Java Web Start* that evaluates the application configuration and are able to install a proper runtime environment on-the-fly [383], they are typically limited to a certain type of application executable, e.g. Java byte code. Currently, there is no general execution engine that is designed to support various types of application executables and is suitable for the employment in a cross-platform operating scenario, as described in Chapter 3.

4.5.1.2 Features

The *Applicationt Execution Engine* aims at running a cross-platform execution environment in which tasks can be deployed and executed in a self-managing way without particular user intervention. The major features are as follows.

Self-Managing Assembly Selection. By dynamically evaluating the current platform configuration and application runtime requirements, the crosslet engine selects suitable code assemblies when an application is to be launched in a self-managing way. An execution policy specifies the overall behaviour of the application execution engine, e.g. always to retrieve the latest code components from remote code repositories or to favor native execution code upon interpreted byte code.

Adaptive Task Processing. In traditional usage scenarios, a user installs a certain application on a computing system in advance to process a task, e.g. MS Winword for editing text documents. By using the application execution engine, the user is no longer specifying which application is to be started but which task he wants to process. This enables the adaptive customization of the task processing on a per platform basis, e.g. switching to different text editors in nomadic usage scenarios.

Extensible Executable Support. While the application execution engine is primarily designed to handle standard Java executables, it can be extended to support particular cross-platform application types as well, e.g. Perl scripts, OSGI bundles or Java Servlets. A specific runtime handler has to be added that knows how to configure the runtime environment, assemble the application and start the execution, e.g. adding a Java wrapper for configuring and controlling a Perl interpreter.

Dynamic Runtime Installation. If an application cannot be run on a given computing system since none of the available runtime environments can be used, the crosslet engine tries to retrieve and configure a suitable one, e.g. by downloading and starting a Java servlet container for running a Java servlet. The runtime installation itself is performed as for regular crosslet applications; appropriate code assemblies are selected and code dependencies resolved without user interaction.

Native System Integration. By using the Java activation library, the application execution engine is able to register crosslet applications to be launched when the user attempts to open a file using the operating system. For example, by double-clicking on a XAR file on the user's desktop, the contained crosslet application is automatically loaded and executed. This feature allows to integrate any crosslet applications into the operating system; acting like a cross-platform application installer.

4.5.1.3 Approach

In an on-demand computing scenario, a user is interested in processing a task without having the need to select and configure suitable applications for the currently employed computing system. In the following approach, an *application execution engine* is presented that receives task descriptions from the user and dynamically customizes the target computing system to run the task without particular user intervention, as shown Figure 4.41.

The task description contains the configuration information of the task, e.g. the type and location of the task input data and the desired task action. The command mapping specifies the application for processing a task and has been configured by an application installer, the user or the system administrator. The crosslet repository is used to retrieve missing application crosslets if a yet not installed application should be launched for the processing of a given task. The application execution engine manages the task execution and prepares customized application environments for running related applications in a self-managing way.

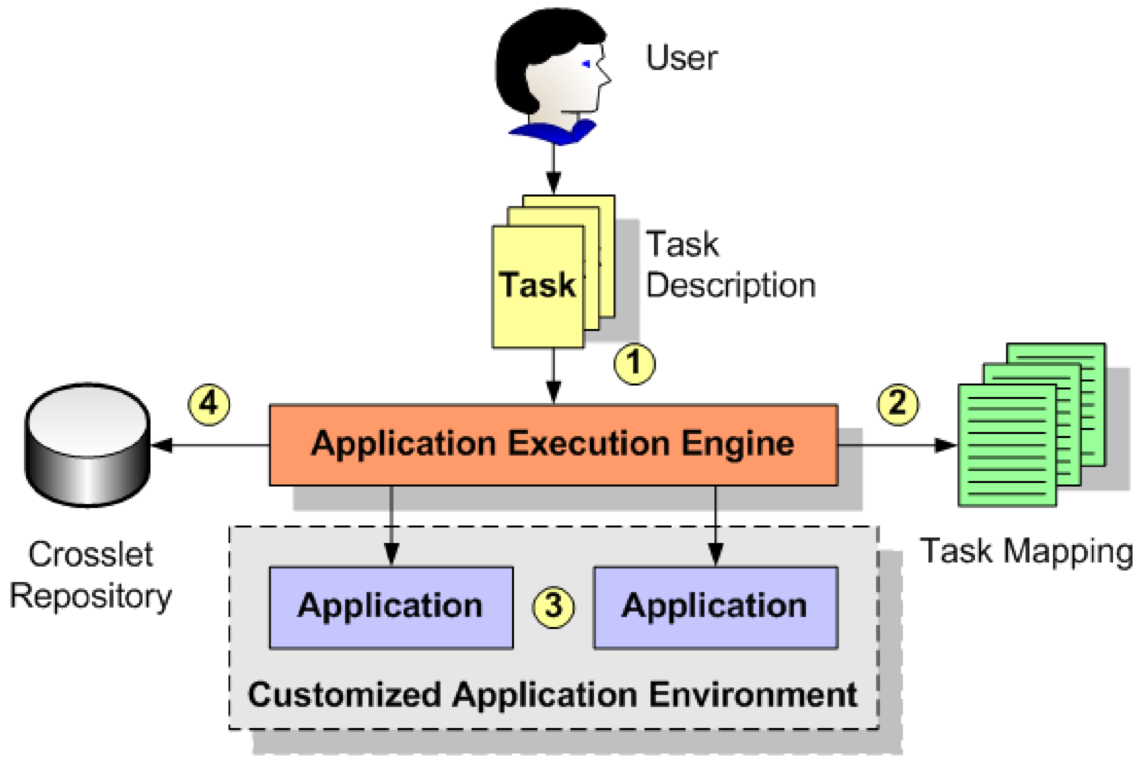


Figure 4.41: Application Execution Engine

For example, if a user wants to print a document with a certain document format, a task description specifies the document location, its content type and a print command object (1). The application execution engine evaluates the received task description and tries to determine matching applications by checking the command mapping entries (2). Afterwards, a suitable application environment is created to run the application (3) and, if needed, related application crosslets are installed on-the-fly (4). As a result, the application execution engine separates the task processing from the application execution. A user can delegate a task to a computing system and does not need to know how to set up and run the required application.

4.5.1.4 Realization

The Java realization of the approach divides the application execution engine in task manager, self-managing application launcher and runtime manager, as shown in Figure 4.42.

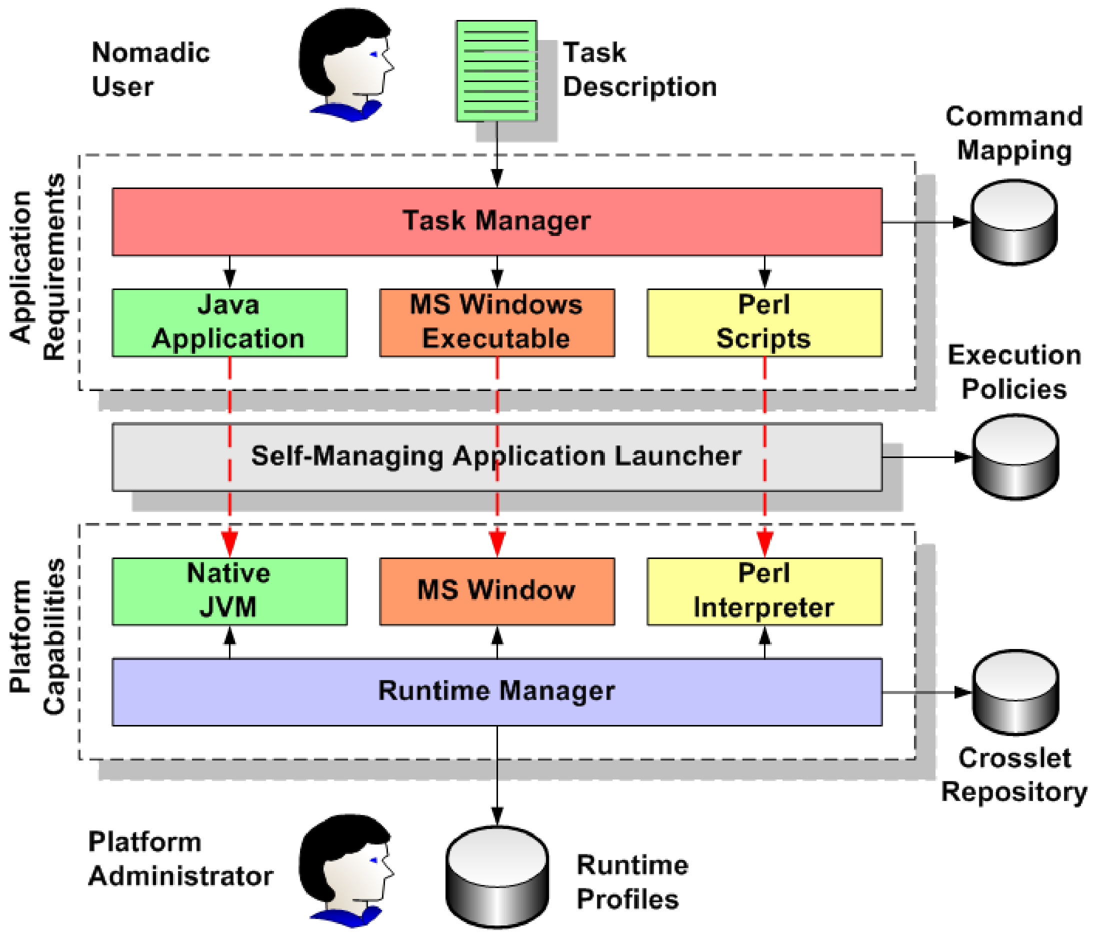


Figure 4.42: Self-Managing Application Launcher

Task Description. The task description is an XML configuration file that is passed to the task manager for processing a given task, e.g. by a nomadic user who wants to edit a specific document. An excerpt of a task description is shown in Figure 4.43.

```
<task-description>
  <properties>
    <property name="command" value="edit" />
    <property name="input" value="/home/spaal/documents/sample.txt" />
    <property name="content-type" value="text/plain" />
  </properties>
</task-description>
```

Figure 4.43: Excerpt of a Task Description

The property `command` indicates the task command, e.g. that the user wants to edit a document. The property `content-type` specifies the document type for selecting an appropriate application to edit the document given in property `input`. More properties may be specified depending on the task type, e.g. passing the location of an output file for writing the results of a computational task.

Command Mapping. The task description is passed to the task manager that resolves a suitable application by searching a matching command mapping like the example shown in Figure 4.44.

```
<command-mapping>

  <activation>
    <properties>
      <property name="title" value="ODIX Text Editor" />
      <property name="content-type" value="text/plain" />
      <property name="command" value="edit" />
      <property name="launcher.id" value="{9E5938EA}" />
    </properties>
  </activation>

</command-mapping>
```

Figure 4.44: Command Mapping

Command mappings are managed using the *Java Activation Framework (JAF)* and configured by an application during installation or by the user during runtime. In the example, an activation configuration is set up for handling the command `edit` and content-type `text/plain`. Instead of directly referring a certain application configuration, a launcher configuration is resolved in the next step user the property `launcher.id`.

Launch Configuration. The idea is to define a common configuration of application and runtime settings, as shown in Figure 4.45, which can be multiply referred, e.g. in distinct menu items.

```
<launch-configuration id=”{9E5938EA}">

  <application>
    <properties>
      <property name="name" value="odix-text-editor" />
      <property name="release" value="1.0+" />
    </properties>
  </application>

  <runtime>
    <properties>
      <property name=”type” value=”native-java” />
      <property name=”execution” value=”process” />
    </properties>
  </runtime>

</launch-configuration>
```

Figure 4.45: Launch Configuration

A custom set of application properties is used to query a matching application configuration, e.g. an application with the name `odix-text-editor` and a release `1.0` or newer. The same is valid for the runtime properties, e.g. using a native JVM and executing the application in a separate process.

Application Configuration. An application configuration defines the properties, the parameters and the dependencies of an application. It is typically defined by the application developer. An example for a legacy Java application is shown in Figure 4.46.

```
<application-configuration id="{93FD9D0E}">

  <properties>
    <property name="name" value="odix-text-editor" />
    <property name="release" value="1.0" />
  </properties>

  <parameters>
    <parameter name="main-class" value="org.crossware.editor.Main" />
  </parameters>

  <dependencies>
    <runtime-profile>
      <property name="type" value="native-java" />
      <property name="release" value="1.4.2+" />
      <property name="execution" value="process" />
    </runtime-profile>

    <collections>
      <collection id="gui">
        <property name="release" value="1.1.0" />
      </collection>
      <collection id="spelling">
        <property name="release" value="1.2.1" />
      </collection>
    </collections>
  </dependencies>

</application-configuration>
```

Figure 4.46: Application Configuration of a Legacy Java Application

The properties section is evaluated when querying an application, e.g. by `name` and `release`. The parameters are passed to the runtime environment for launching the application, e.g. the `main-class` of a Java application. Finally, the dependencies section specifies the application requirements, e.g. the properties for querying a suitable runtime environment and the Java class collections to configure the CLASSPATH.

Runtime Profile. The platform-specific runtime profiles are set up by the platform administrator and contain information about how to initialize a certain application environment on the current computing system, e.g. where to find the PERL interpreter or how to set up the environment settings such as the CLASSPATH for a legacy Java application. An example of a runtime profile configuration for launching a JVM in a separate process is shown in Figure 4.47.

```
<runtime-profile id="{8A750732}">
  <properties>
    <property name="type" value="native-java" />
    <property name="execution" value="process" />
    <property name="version" value="1.4.2" />
    <property name="command" value="/usr/sdk/sun-jdk-1.4.2/bin/java" />
  </properties>
</runtime-profile>
```

Figure 4.47: Runtime Profile Provided by the Platform Administrator

In the example, the properties indicate that this configuration is related to an application environment that launches a native JVM in a separate process. It corresponds to Sun JRE 1.4.2 and is started by passing the command-line `/usr/sdk/sun-jdk-1.4.2/bin/java` to the shell.

Runtime Plugins. To prepare particular application environments, extra runtime plugins may be added to the application execution engine to evaluate the related settings in the application configuration and runtime profile. For the example above, an excerpt of the related runtime plugin code is shown in Figure 4.48.

```
// retrieving the main class and class path
// from the application configuration
String mainCls = applicationConfig.getMainClass();
String classPath = applicationConfig.getClassPath();

// creating the shell command and starting the process
String shellCmd = runtimeProfile.getCommand() + " " + mainCls;
String[] env = new String[] {"CLASSPATH = " + classPath};
Process proc = Runtime.getRuntime().exec(shellCmd, env);

// redirecting the stdin/stdout/stderr of the application environment
IConsole console = CConsoleManager.getConsoleManager().createConsole();
console.setIn(proc.getOutputStream());
console.setOut(proc.getInputStream());
console.setErr(proc.getErrorStream());
```

Figure 4.48: Runtime Plugin for Launching a JVM in a Separate Process

The plugin gets the application configuration and runtime profile for initializing the application environment. The main class and class path of the Java application are read and the shell command for starting the process is assembled. The location of the Java tool is retrieved from the runtime profile shown in Figure 4.47. A new process is started with `exec` and the Java command line is passed to the shell as well as the environment settings. Finally, the plugin redirects the input, output and error streams to a separate console that for instance may be forwarded to a remote terminal. Beside this described scenario for hosting a legacy Java application, further application environments may be created provided there is a suitable runtime plugin and runtime profile available on the currently employed computing system.

4.5.1.5 Autonomic Application

The application execution engine approach can be used to implement *Autonomic Task Deployment* by on-demand launching of related task processing applications in a cross-platform operating environment, as shown in Figure 4.49.

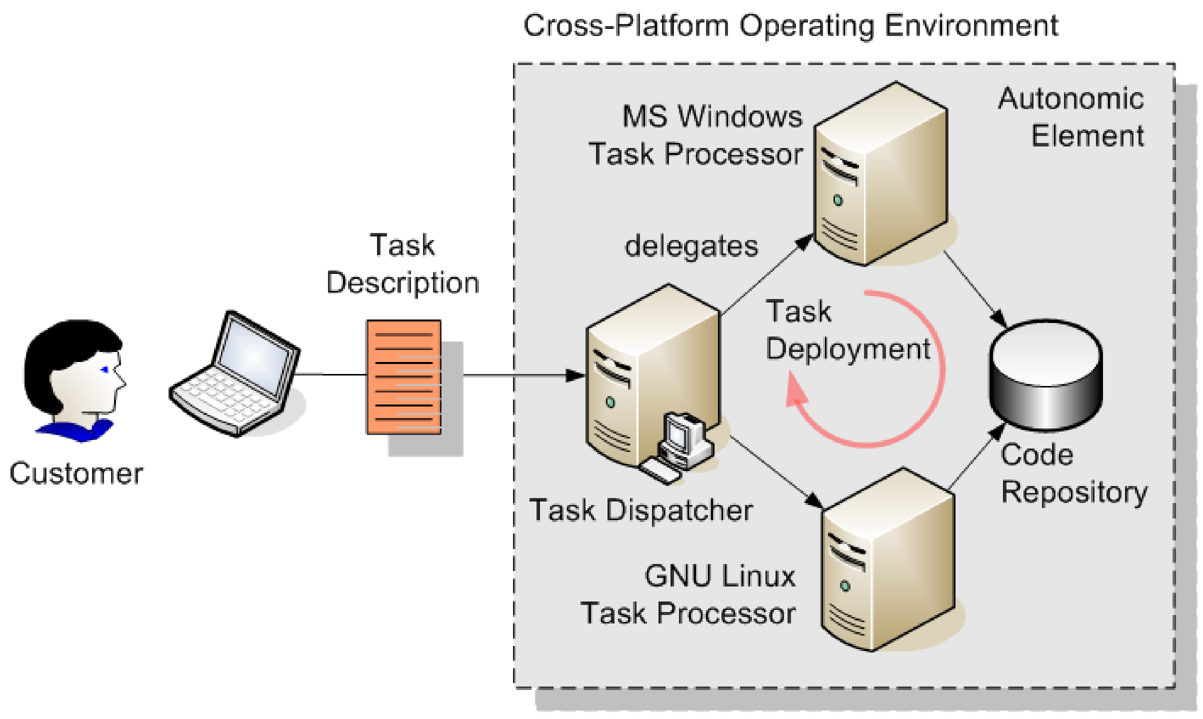


Figure 4.49: Autonomic Task Deployment

A task processor runs an application execution engine and is able to process customs tasks by interpreting a task description, e.g. for retrieving a media file from a content repository and performing certain media indexing operations. Before a task is actually processed, the task processor evaluates the task description for required crosslets and may download missing ones from a shared code repository. The task dispatcher represents the single point of command and control for delegating received tasks to dynamically discovered task processors, e.g. in a peer-to-peer network.

The autonomic operation is performed by monitoring and controlling the tasks running on the task processors. From this point of view, the customer's computer is the requesting element, the task dispatcher acts as the autonomic manager, the task processors are the managed elements that altogether form the cross-platform operating environment as the autonomic element. The customer is not aware on that task processor his or her task is actually processed and how required resources are appropriately configured. In contrast, he or she gets the illusion of an ever ready-to-use cross-platform operating environment for random computing tasks. For example, if a customer wants a computing task to be processed, he or she creates a task description and passes it to the task dispatcher. Depending on the computing task, available task processors and code assemblies, the dispatcher delegates the task, e.g. to a MS Windows computing systems if the task can be only processed by a native MS Windows application *(self-configuration)*. Since there is an ever-changing task deployment scenario, the task dispatcher checks the computing load and may adapt the deployment of new tasks according to a given policy, e.g. for ensuring a minimal task processing time *(self-optimization)*. Another monitoring issue is the tracking of computing progress and results. In case a task could not be processed, e.g. due to an unexpected shut down of a task processor, the task dispatcher can redeploy the task to another task processor and restart the computing *(self-healing)*. If the task dispatcher is notified about an issued shut down request, it could also migrate an existing task and its result to another task processor [see Section 4.7]. The same is valid for protecting a single task processor to be overloaded due to an unexpected lengthy task execution *(self-protection)*.

4.5.1.6 Related Work

The interconnection of distributed computing systems has made it possible to easily deploy applications to a large number of computing systems on-demand. A related request is the spontaneous launching of random applications not yet known when the target computing system has been set up. In this context, the provision and configuration of a suitable application environment to run the deployed application is a non-trivial task. In the following overview, related approaches and exemplary solutions towards *on-demand application execution* are categorized and compared against the application execution engine approach.

Native Application Environment. In the first category, related solutions address the need to deploy and run *native application code*. For example, if legacy standalone applications are to be reused to parallelize the computation of customer tasks and to utilize idle workstations. In *Butler* [255], a machine registry is introduced to list all connected machines in a local network. If an application is to be executed on a remote computing system, the registry is queried for a suitable machine and an appropriate execution environment is prepared, e.g. copying required shared libraries to the target machine. The actual application execution, however, is not altered. In fact, an application can be only executed if there is a workstation with a CPU capable to run its machine code. A related solution for *volunteer computing* is *Berkeley Open Infrastructure for Network Computing (BOINC)* that offers to deploy and run applications on idle workstations in the Internet [32]. In contrast to Butler, the user of an Internet computing system first decides to support a specific distributed computing project and then requests its BOINC client to install a related computing application on his or her workstation. An application is not pushed to remote computing systems but pulled from a central project repository to the local computing system. Popular projects using BOINC are *SETI@Home* and *Einstein@Home* [99]. Besides supporting volunteer computing projects over the Internet, BOINC can also be used to run a desktop Grid computing system in an enterprise network. Actually, any standalone application can be adapted to run within a BOINC application environment. Due to platform-specific code assemblies, however, there is no or only little support of dynamic component deployment and application composition, e.g. for launching a suitable task processor not yet installed on the target computing system.

System Virtual Machine. The next category is related to on-demand computing solutions based on launching a *System Virtual Machine (SVM)*, e.g. *VMWare Workstation* or *Sun Virtual Box*. A computer hardware environment is virtualized and a guest operating system is bootstrapped to run the requested application. This approach allows running native application code in a sandbox without interfering with the host system or applications running in another SVM. With the advent of *cloud computing*, this solution is widely adopted for deploying legacy computing applications in a shared computing infrastructures [220]. For example, *Amazon EC2* supports the creation of virtual machine images that can be used to easily set up a large number of application installations in the Amazon cloud [9] and support remote task processing. While the required application environment is dynamically prepared, a virtual machine image cannot be modified without launching it, e.g. if a contained application is to be updated. Further, distinct application services running in the same virtual machine may affect each other, e.g. in an application service provisioning scenario. A possible solution is a *service on-demand architecture (SODA)* for application service hosting utility platforms [186]. Application services are isolated in different virtual machines and separately launched on-demand. A common service switch is installed to direct client requests to appropriate application services. For wide-area distributed computing, the SVM approach offers the dynamic deployment and secure execution of untrusted application, as in the *XenoServer* project [124]. Related execution platforms are deployed around the globe and offer the multi-customer hosting of application services in distinct virtual machines on-demand. In this scenario, a particular issue is the efficient management of globally distributed computing resources. In *Oceano*, an autonomous management system is implemented to manage global utility computing resources [145]. Automated resource demand-supply control systems support the automated service deployment and dynamic capacity sizing of services. In a wide-area deployment scenario, however, a common problem of SVM-based solutions is the initial setup of the SVM installation which usually requires some administration knowledge.

Process Virtual Machine. In a *Process Virtual Machine (PVM)*, portable code is run by an interpreter in a regular process environment of the host system [see Section 3.2.1]. The overall resource consumption of a PVM is smaller than of a SVM and the startup time is usually much faster. In conjunction with the simple and instantaneous code deployment over the Internet, this has particularly driven the development of *Rich Internet applications (RIA)* [216] e.g. applets written in *Adobe Flash* and *Sun Java*. In this context, the web browser acts as an application container and dynamically installs an appropriate runtime environment to launch the selected RIA, similar to the Application Execution Engine. A common disadvantage is the focus on particularly implemented and self-contained web-based applications. Besides running a RIA in a browser environment, so called Rich Client Applications may be deployed over the Internet and run in a separate process environment. For example, regular Java applications may be launched on virtual any Internet computer system on-demand by using *Sun Java Web Start* [383]. An application description, the *Java Network Launch Protocol file (JNLP)*, is downloaded and dynamically evaluated for installing the appropriate JVM version and launching the Java application. In fact, Sun's Java Web Start actually represents the most adopted PVM solution for deploying and running regular Java desktop applications on-demand. Of course, there are further PVM implementations, e.g. running Python or Perl scripts, but they lack support for the dynamic installation of the required runtime environments in an Internet computing scenario. Once a PVM-based application container is installed, there are various approaches to run user applications on-demand and to support task processing. In *Argos*, an *extensible personal application server* allows deploying and composing Java applications on desktop computers by introducing a custom application description [249]. In *Hydra*, the application container supports the policy-based aggregation of Java applications from mobile code components travelling between computers [313]. For remote and distributed task processing with administered application containers, there are popular PVM-based solutions for regular applications like *Google's AppEngine* [143], for standardized JEE servlets like *Snap* [131] and for proprietary applications like

DistrIT [88]. While PVM application containers easily support on-demand application execution, a common issue is the concurrent processing of multiple tasks within the same virtual machine. As an example, if one task processing application fails, concurrently hosted applications may be affected as well.

To summarize, there are many application containers supporting *on-demand application execution*. Some of them support single application execution like Java Web Start while others have been designed to enable multi application execution like BOINC. Native application environments like Butler promise the best computing performance but lack on-demand application deployment in a cross-platform operating scenario. The SVM solutions like VMWare Server can be used to run prepared virtual images in isolated process environments and on a broad range of heterogeneous computing platforms. A basic issue of SVM solutions, however, is the overhead to virtualize a legacy hardware environment and to provide a customized application installation, e.g. an offline virtual image cannot be modified and the hot deployment of large virtual images is not feasible over the Internet. An alternative approach is the use of a PVM and running customer applications in legacy processes. The startup time is typically less than the one of a SVM, the resource requirements are moderate and application executables may be easily deployed over the Internet. From this point of view, SVM-based approaches enable *heavy-weight application execution* while PVM-based approaches make *light-weight application execution* possible. A common limitation of related solutions is the focus on a single application container implementation and type of runtime environment, e.g. Sun Java Web Start can be only used to deploy and run Java applications. The presented *application execution engine* is divided in a task manager and a runtime manager. If an application is requested, a suitable application configuration and runtime installation are negotiated by a self-managing application launcher, the application components are retrieved from the crosslet repository and finally the application is launched. Various Java applications may be concurrently hosted in a shared JVM or separately executed like regular native executables. As a result, the application execution engine does not only control the application execution but also the runtime configuration.

4.5.2 Roaming User Profiles

The seamless customization of distributed computing systems is needed to provide the illusion of a pervasive application environment to the nomadic user. In this section, the conceptual approach of roaming user profiles is described for customizing the application installation by considering system, application and user settings. The realization is illustrated and the application of the approach for deducing missing profile settings in a self-managing way is outlined.

4.5.2.1 Motivation

To work with a software application, a user typically customizes the application configuration to suit his or her preferences, e.g. by adjusting the visual appearance and removing unused menu items. The goal is to ease the handling of the application and to store user-related configurations like the mail account settings for the next application session. While some applications put the settings in separated configuration files, e.g. in their installation directory, more and more applications use shared user profile repositories, e.g. managed by an enterprise LDAP server. They can be centrally administered and also be used to synchronize distributed profiles located on alternately employed computing systems. At best, a nomadic user gets the illusion of a pervasive application environment that is not bound to a certain computing system but is restored on each one he or she is currently working at, e.g. across various computing systems in the enterprise.

In Java, legacy support for storing user profiles is provided by the Java preferences package `java.utils.prefs` that usually backups the settings in a configurable native profile repository, e.g. the MS Windows registry or a remote profile database. While this approach is feasible for single-user application environments, it has a major drawback in a multi-user and multi-tasking scenario. The original preferences approach has been designed to manage system and user profiles that are associated with the user account having launched the JVM. In a shared JVM, however, various tasks are concurrently run by different user accounts and there is a particular need to offer each transparent access on private application and user profile settings. Another issue is the dynamic adaptation of roaming user profiles due to different configuration setups found in a cross-platform operating scenario, e.g. an application is to be used on a computing system where it has been originally not installed; hence application-specific profile settings are missing in the profile repository.

4.5.2.2 Features

The overall goal of the *Roaming User Profile* approach is the seamless provision, synchronization and customization of user preferences to provide the illusion of a pervasive application environment to a nomadic user. The major features are as follows.

Roaming Profile Synchronization. A user customizes his or her computing environment by modifying various configurations that are stored in a user profile, e.g. in the user's home directory. The XDK supports the nomadic synchronization of this profile with a remote master copy; similar to the MS Windows domain user account approach. As a result, a nomadic user can switch to distinct computing systems that are customized according to his or her user profile.

Multi-Session Profile Handling. While the legacy JRE offers a standardized way to store system and user preferences in the local profile, the related implementation has been designed for a single-session JVM only. The XDK approach extends the standard preferences package and enables transparent support of multi-session profile handling using the Java standard. Applications running in different sessions can access related user preferences without particular implementation.

Self-Managing Profile Deduction. Another feature of roaming user profiles is the self-managing deduction of missing preferences by evaluating profiles of other users. For example, a nomadic user might not know which proxy server to use for passing the firewall on a new computing system. However, a former user has already configured the right proxy and left the settings in the local profile storage. The XDK applies this setting and completes the current user profile on-the-fly.

User Document Repository. In addition to custom user preferences, the computing environment of a user also encompasses the document files the user works on. Since the XDK is run by the same system user but may host various sessions of distinct nomadic users, a particular home directory for each session is created and the document files therein are synchronized with a user document repository. The user gets the illusion of travelling with his or her documents while being on the move.

Legacy Application Support. If a Java application is started in a separate JVM, the default working directory is the user's home from where the preferences are loaded as well. A related XDK feature is the support of legacy applications for nomadic use by extending the regular JVM with extra preference classes and providing transparent access to the synchronized user profile and documents. A legacy application does not need to be aware of this and can still use standard JRE methods.

4.5.2.3 Approach

The basic idea of the approach is to synchronize the user preferences on multiple computing systems by introducing remote profile repositories and roaming user profiles, as illustrated in Figure 4.50.

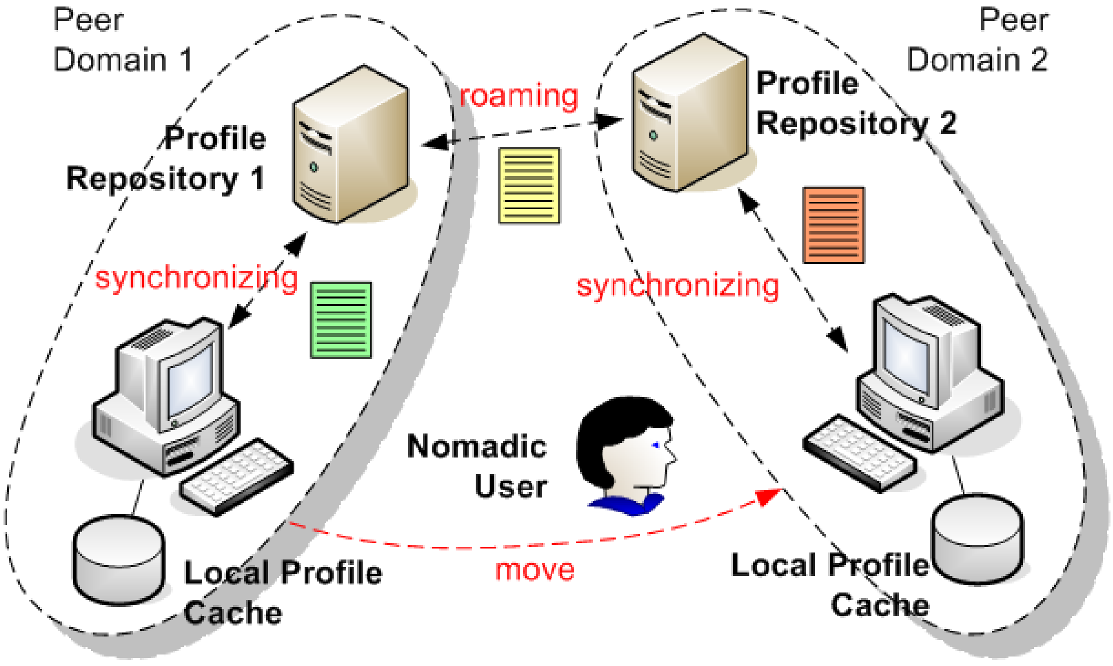


Figure 4.50: Roaming User Profiles

If a user customizes the application environment on a computing system, the user preferences are stored in a local profile cache and reloaded the next time the user logs in. In addition, the preferences are synchronized with a remote profile repository in the same peer domain, e.g. the Intranet of an enterprise or a pre-defined group of computing devices. Assuming that the profile repository is reachable by all peers in the peer domain, a user may alternately use distinct peer computing systems and gets his or her user preferences restored. In case a nomadic user moves to another peer domain with a different profile repository, e.g. from peer domain 1 to peer domain 2, the profile repository of domain 2 requests the user profile from repository 1 and becomes the new synchronizing profile repository for the current user session. This speeds up further synchronization of the user profile in the new peer domain and allows administrators to connect all peers of a peer domain to the same profile repository. By partitioning the user profile in static and roaming fragments, platform-specific user preferences are not synchronized that allows users to refine the application environment on a per-platform basis, e.g. configuring different display resolutions on various computing devices. Moreover, user documents in the local working directory are also synchronized and restored with the user profile. For regular Java applications running in a multi-session runtime scenario, the respective user preferences and documents are transparently provided and there is no need to adapt the application preferences handling.

4.5.2.4 Realization

The realization of the approach introduces a multi-session profile manager that handles the profile synchronization with the profile repository, as shown in Figure 4.51.

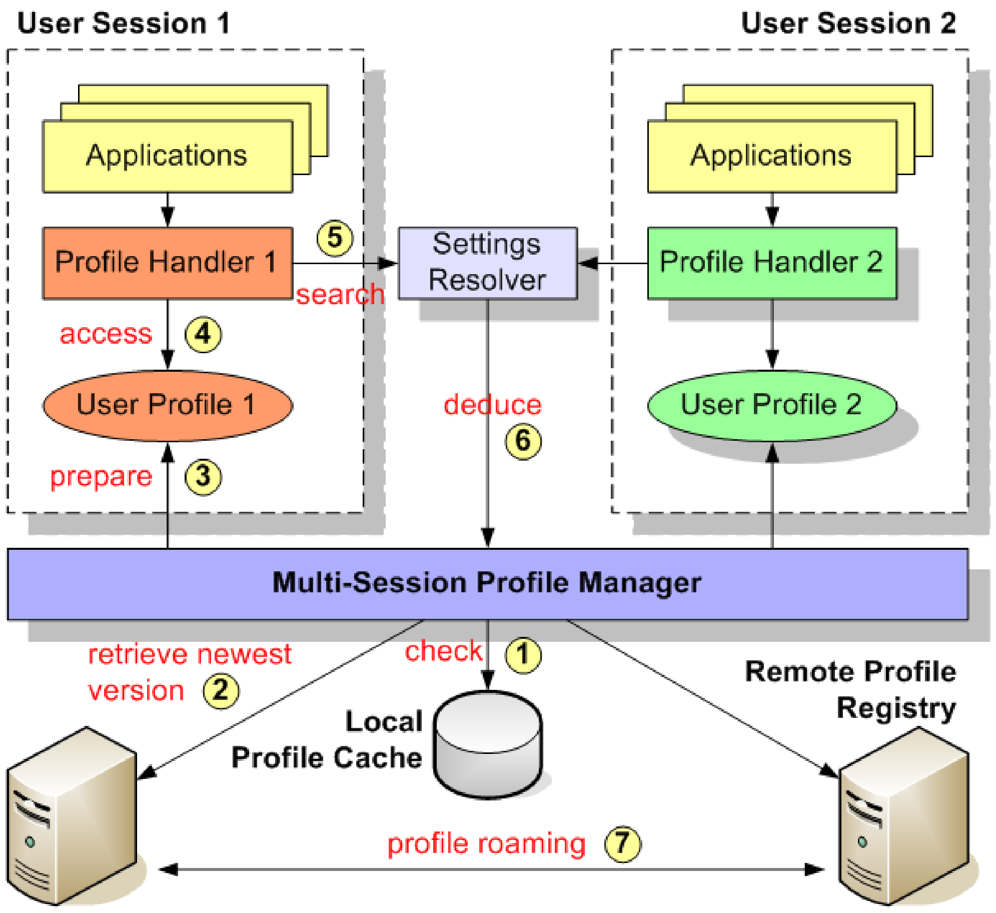


Figure 4.51: Multi-Session Profile Manager

If a new user session is started, the profile manager checks the local profile cache (1) and the remote profile repository for the newest release of the related user profile (2) by comparing the serial numbers. If needed, a newer user profile is retrieved from the profile registry and stored in the local profile cache. Then, a user profile copy is prepared by the profile manager (3) that can be accessed by the profile handler of the user session (4). The profile handler provides a transparent interface to the user profile and can also complete missing profile settings by using a setting resolver (5). For example, if a user has installed a new application that needs information about the local network configuration, the settings resolver can be used to evaluate the system profile and other user's profile for an appropriate setting (6), e.g. determining the last working configuration of a web proxy for the current computing system. Finally, user profiles may be stored in regular directory systems like an LDAP server that offer built-in support for profile roaming (7).

Profile Loading. The profile settings are organized in a hierarchical structure and can be easily explored and read, as shown in Figure 4.52.

```
// determine the principal id of the session
CPrincipalId prpId = sessionManager.getSession().getPrincipalId();

// load the profile from the profile repository
IProfile profile = profileRepository.loadProfile(prpId);

// create the path to the application setting
String key = "application/editor/textpad/1.0/font";

// read the setting
Font font = (Font) profile.getValue(key);
profile.setValue(new Font(font.getName(), font.getStyle(), 10);

// commit the changes of the profiles
profile.commit();
```

Figure 4.52: Explicit Profile Handling

First, the principal id of the current session is determined and used to load the profile of the session user. Then, an application specific key is created to read the associated application setting. In this example, a font setting is read, its font size is set to 10 and written back to the profile. Finally, the profile is committed and synchronized with the profile repository.

Native Preferences Handling. The explicit profile handling allows developers to determine the user session and access the respective profile. However, existing Java applications using legacy preferences handling are not aware of user profiles and use a different approach shown in Figure 4.53.

```
String fontName = Preferences.userRoot().get(key);
```

Figure 4.53: Legacy Preferences Handling

In contrast to the profile repository implementation, the legacy preference handler does not support multi-session operation and thus the hosting of multiple Java applications in the same JVM may cause problems in access the respective user profile. A remedy is offered by the JVM by installing a custom preferences handler, as shown in Figure 4.54.

```
-Djava.util.prefs.PreferencesFactory =
                    org.crossware.xdk.profile.CMultiSessionProfileFactory
```

Figure 4.54: Configuration of Custom Preferences Handler

The environment setting `java.util.prefs.PreferencesFactory` replaces the built-in preferences handler with a customizable implementation that can be used to support legacy preferences handling in a multi-session environment.

Multi-Session Preferences Implementation. A custom preferences implementation has to extend the class `java.util.prefs.AbstractPreferences` and can implement various methods of the service provider interface (SPI) to control the preferences handling. An excerpt of the implementation is shown in Figure 4.55.

```
public class CMultiSessionPreferences extends AbstractPreferences
{
  // a table of session profiles
  private Map<CSessionId, IProfile> m_profiles =
                              new Hashtable<CSessionId, IProfile>();

  // retrieve the session profile by determining current thread group
  private synchronized IProfile getProfile()
  {
    ThreadGroup tg = Thread.currentThread().getThreadGroup();
    CSessionId sessionId = sessionManager.getSessionId(tg);

    IProfile profile = m_profiles.get(sessionId);
    if (profile == null)
    {
       profile = repository.loadProfile(userId);
       m_profiles.put(sessionId, profile);
    }

    return profile;
  }

  // overwrite the service provider interface for reading a value
  protected synchronized String getSpi(String key)
  {
    return getProfile().getValue(key);
  }
}
```

Figure 4.55: Multi-Session Preferences Implementation

When the application requests access to the user preferences, as shown in Figure 4.53, the registered preferences factory returns an instance of the multi-session preferences implementation. An application method call of `public String get(String key)` is delegated to the method `protected String getSpi(String key)` shown in Figure 4.55. The method `getProfile` is called to retrieve the current session that is associated with the thread group of the task execution. If the respective user profile has not yet been loaded, the profile repository gets the profile from the profile cache. Finally, the requested setting is read from the user profile and returned to the calling application. This is performed without modifying the application and can also be used for customizing the preferences handling of legacy Java applications launched in a separate JVM process. In this context, the synchronization of document files for a certain session is similarly done by using the environment setting `user.home` that denotes the working directory to be used by the JVM process. Before the legacy application is launched, the user's files are copied to the working directory and when the session ends, the documents are zipped and synchronized as part of the user profile.

4.5.2.5 Autonomic Application

The Roaming User Profile approach can be used to implement *Autonomic Environment Customization* for nomadic application scenarios with heterogeneous computing systems involved, as shown in Figure 4.56.

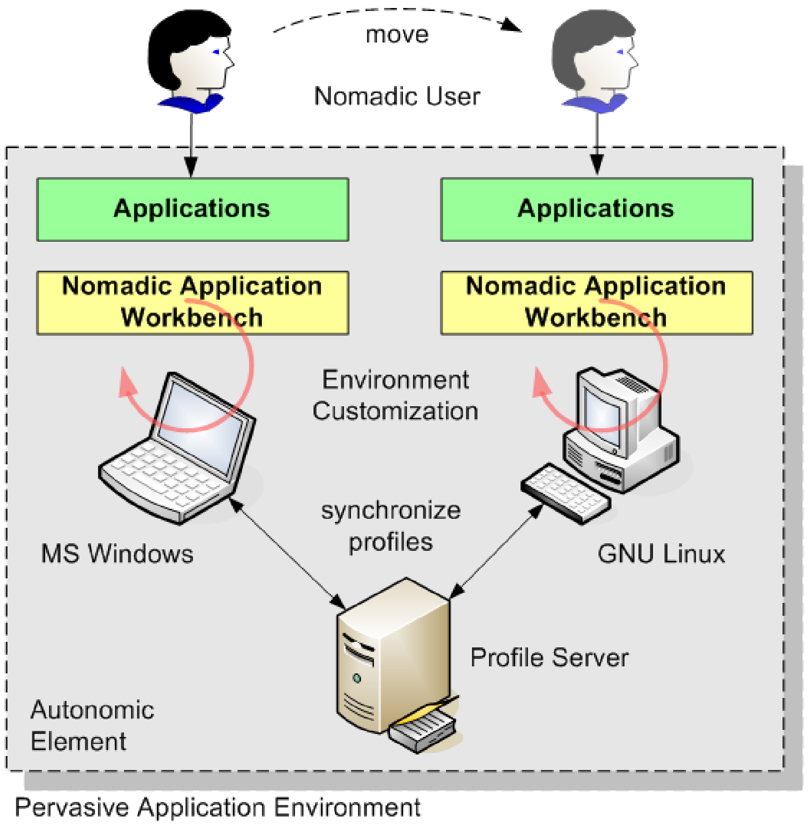

move
Nomadic User
Applications
Applications
Nomadic Application Workbench
Nomadic Application Workbench
Environment Customization
MS Windows
synchronize profiles
GNU Linux
Profile Server
Autonomic Element
Pervasive Application Environment

Figure 4.56: Autonomic Environment Customization

A nomadic user moves from one computing system to another and wants to run his or her applications with the same profile settings. For this purpose, a nomadic application workbench [see Section 5.3] is used to prepare a suitable application environment and a profile server is introduced to synchronize the customized user settings across distinct computing systems. Along with the dynamic deployment of the user's applications, this results in a pervasive application environment in which the different capabilities and configurations of heterogeneous computing systems are hidden from the user. For example, if an application exists as MS Windows and GNU Linux variant, the nomadic application workbench will seamlessly customize the environment and restore the user's profile before the appropriate variant is launched.

The nomadic application workbench makes autonomic operation possible by controlling the user profile synchronization and monitoring its customization later on. From this point of view, the hosted applications are the requesting elements, the workbench acts as the autonomic manager, the platform configurations are the managed elements and the pervasive application environment represents the autonomic element. The user and the applications are not aware of the profile synchronization in the background and get the illusion of a seamlessly roaming user profile. For example, if the user logs into a new computing system, the workbench determines the related profile server, synchronizes the local profile cache and customizes the workbench without user or application intervention, e.g. launching applications configured in the autostart section *(self-configuration)*. In this context, a particular issue is the transmission of the user profile from and to the master profile server. The implementation supports the configuration of multiple profile servers and the ability to move the master copy to the nearest profile server, e.g. roaming the user profile to a profile server in the local network for shortening the synchronization time *(self-optimization)*. Once the user has installed his or her applications, the related installation configurations are stored in the user profile and also synchronized. In case a configured application has been not yet deployed on the currently employed computing system, the workbench automatically installs the application according to a given policy, e.g. when the user logs in, on first access or always in the background *(self-healing)*. Since the nomadic application workbench is designed to support multi-session operation, a further issue is the protection of user profiles against access and manipulation by other users. This is achieved by the multi-session profile manager that denies unauthorized access during runtime and encrypts the user profiles before storing them in the local profile cache. In turn, the signature of the user profile is checked during login and before evaluating the profile settings *(self-protection)*.

4.5.2.6 Related Work

The shared and alternate use of various computing systems by multiple users creates individual application environments. A particular user request is enabling ubiquitous personalized computing by adjusting the current application environment according to preferences of the user [387]. A common approach to save and restore the related settings is the introduction of various profile repositories, e.g. for managing system, application and user profiles. In the following overview, static and dynamic profile approaches that can be used to create the illusion of a *pervasive application environment* are outlined and evaluated for implementing roaming user profiles in Java.

Static User Profiles. A common strategy to manage user settings is to use *static user profiles* that are not modified by the profile repository once they have been stored. In a simple implementation, text files are used to store the profile settings as *key-value pairs*. Popular examples are so called *INI-files* that have been widely used in MS Windows 3.x and can be still found in particular scenarios, e.g. for configuring available plugins in an application framework. The basic advantage is the straightforward processing of the configuration files and the simple handling in a distributed computing scenario, e.g. the configuration files can be downloaded from a central profile server and uploaded if changes have to be committed. The major drawback is the scattered organization in a multi-application environment, e.g. every application uses a separate configuration file, introduce various profile definitions, the files may be modified or corrupted by accident. An advanced approach is the introduction of *profile registries* like in MS Windows 95 and later. All configuration settings are stored in the registry and there is an official registry specification where settings are to be stored, e.g. user and machine settings are stored in different *hives* [239]. For Java applications, the actual profile handling is shielded by the *Java Preferences API* that transparently maps profile operations to the underlying profile registry of the hosting platform [344]. Concerning roaming user profiles, a common approach is the synchronization of a local profile cache with the remote profile repository, e.g. a server-based user profile is downloaded to the local registry upon login to the workstation [384]. This happens in the same way for every workstation and the profile settings are usually not adapted to platform-specific configurations. While this also works for Java applications in uniform computing environments, e.g. MS Windows workstations connected to the same domain controller, this approach fails for heterogeneous computing systems. In fact, on-demand Java applications that alternately run on MS Windows and Linux cannot use the standard profile storage approach. Another issue of Java preferences is the all-or-nothing synchronization scheme that is not feasible for profile management over the Internet.

Dynamic User Profiles. A remedy to the roaming issues of static user profiles are *dynamic user profiles* that are not completely synchronized in advance but allows accessing selected profile settings by request. A common approach towards dynamic user profiles is the provision of *directory services*. For MS Windows, the *MS Active Directory Service* supports remote access to dedicated registry settings. For Linux operating systems, similar directory service implementations are used that typically conforms to the *Light-Weight Directory Access protocol (LDAP)* [205]. For Java applications, a suitable *Java Naming and Directory (JNDI)* adapter can be used to connect to a remote directory service [343]. In a cross-platform computing scenario, remote directory services may be used to bridge application and platform boundaries, e.g. for managing common user settings of MS Windows applications, Linux applications and Java applications in the same profile repository. A common limitation of regular directory services is the static mapping of profile request on profile provision. For example, a platform profile provides the address of a network proxy with a certain profile key while an application requests the address with another profile key. As a consequence of this, a roaming user has to manually map both keys to create a valid configuration that makes it difficult to support on-demand computing in a large-scale cross-platform operating environment [264]. A possible solution to that are the introduction of *semantic user profiles* and the use of semantic matching and reasoning tools as proposed in [325]. This enables the separation of profile repositories operating at a syntactical level from applications evaluating profiles at a semantic level. For mobile computing devices moving across different networks, new forms of dynamic service discovery and integration can be realized. By now, these approaches have not been widely adopted at most due to increased implementation efforts for regular application developers.

In comparison, *static user profiles* are most suitable for the centralized synchronization of user and application settings across uniform computing systems located in an Intranet environment. For cross-platform operating environments with various profile repository installation, *dynamic user profiles* allows roaming of user settings from one site to another, e.g. in a mobile computing scenario. There are advanced and yet not widely adopted approaches like the *semantic user profiles* that address the dynamic mapping of profile requests on profile definitions at a semantic level. For Java applications, the Java Preferences API represents the legacy approach to manage local user profiles. The JNDI binding provides uniform access to remote profile settings and hides the current directory service implementation. From this point of view, the *roaming user profile* implementation benefits from the uniform JNDI binding to synchronize a local profile cache with different types of remote profile registries in self-managing way. Though the semantic mapping of different user profiles is not supported, a customizable settings resolver offers the context-based adaptation of user profiles, e.g. for adjusting the settings of mobile applications according to the platform characteristics like screen resolution and network bandwidth. In contrast to the native Java preferences implementation, an extended profile service provider offers the transparent access on user-specific profiles via the standard API, e.g. for running legacy Java applications in a shared JVM. Moreover, the environment customization is not limited to profile settings but may be configured to handle seamless document synchronization as well. For example, the workplace of a user is managed across distinct computing stations and can be easily accessed by legacy Java applications. As a result, the roaming user profile approach enables *pervasive environment customization* not only for custom but also for legacy Java applications that can be deployed over the Internet.

4.6 Virtual Object Interconnection

In a distributed computing environment, networked applications are executed on distinct computing devices and usually interact with each other by using specific network middleware approaches, e.g. based on CORBA, SOAP or XML-RPC. The network middleware separates the business logic from the network handling needed to communicate with remote parties and frees the application developer from low-level and often proprietary network programming. This is achieved by introducing particular network wrappers, stubs and skeletons, which are usually generated by helper tools during design time. However, once the application is tight to the middleware implementation chosen by the application developer, it cannot be run with a different one, e.g. for dynamically connecting a remote resource using a different network protocol. In Section 4.6.1, a distinct Java object communication approach is introduced, *Java Method Streams*, which features the adaptive remote interconnection of Java objects; especially the selection of appropriate middleware implementations during runtime and self-managing reconnection of interrupted object bindings [162, 273]. In this context, the practice has shown that the hosting of multiple Java applications within the same Java Virtual Machine (JVM) raises new challenges concerning the configuration and control of object interconnections. While regular stubs and skeletons have been originally designed to connect application objects hosted in distinct execution environments, they lack particular support for controlling interconnections of application objects in the same execution environment. In Section 4.6.2, a new approach to encapsulate Java objects in so called *Java Object Spaces* and to control their inbound and outbound communication following given system policies and application configurations is presented, e.g. for implementing a particular object access control and securing application data [278].

4.6.1 Java Method Streams

The interconnection of distributed Java objects is typically performed on the network level using a specific middleware approach. In this section, remote method streams are presented that introduce *virtual object interconnection* by separating virtual object bindings from real object links. The realization using Java Dynamic Proxy is described and the application for establishing and maintaining remote bindings to nomadic objects in a self-managing way is illustrated.

4.6.1.1 Motivation

In object-oriented programming theory, a software application is composed of distinct object instances that are bound by using interface references and communicate with each other by exchanging messages. In practice, this is realized by introducing object references that act as placeholders for the object instances and are used by sender objects to transmit a message via method call to the referred objects. In a local computing environment, the object reference is linked to the referred object instance by using its memory address. For distributed computing scenarios, this is not possible and has led to object middleware approaches that virtually link object reference and object instance by introducing object stub and skeleton objects [370]. Following the *Proxy* pattern [129], they transparently pass the message from the caller to the callee while bridging hosting boundaries, e.g. by using suitable network connections. From this point of view, the application is interested in *virtual object binding* and does usually not want to be bothered with the *real object linking* and eventually involved network communications.

While network middleware allows developers to focus on the business logic and manages network communication issues mostly without application intervention, the drawback is the tight coupling of application and selected middleware implementation; hence the object binding and object linking issues. Consequently, an application is typically not able to bind a remote object without the information about how to link the remote object, e.g. which network protocol and parameters have to be used to establish a connection. Though there are options to customize the object linking on the connection level, e.g. by using particular object registries and self-managing reconnection interceptors as in the CORBA approach [252], they are specific to the current middleware implementation and cannot be applied with different solutions. In turn, the customization of object bindings on the application level, e.g. by introducing access control binding aspects, is not possible for the developer without specifying the middleware to be used for transmitting the message.

4.6.1.2 Features

Similar to the concept of distribution transparency in *RM-ODP* [364], the *Java Remote Method Stream* approach aims at supporting the self-managing operation of local and remote object bindings without customizing the actual business logic [271]. The major features are as follows.

Dynamic Middleware Activation. Java Remote Method Streaming allows developers to select and initialize the actual network middleware during runtime, e.g. by dynamically negotiating the connection type with the remote party and by binding objects using a matching network middleware implementation. In addition, the employed middleware of established object bindings can be altered without breaking the link, e.g. switching from RMI to CORBA due to a changed network scenario.

Custom Binding Composition. Every method call issued from the caller to the callee is passed through the method stream and controlled by chained interceptors [371]. They can be used to add object independent functionalities dynamically to arbitrary objects, similar to the approach introduced by Aspect-Oriented Programming (AOP) [195]. This separates the *streaming logic* defining how a method call is passed from the *business logic* defining how the method call is processed.

Invariant Object Identifier. To register and locate Java objects, invariant object identifiers are introduced that are dynamically resolved and mapped on a specific object address suitable for the network middleware currently in use. By separating *object referencing* from *object addressing*, an application can establish an object connection even if the underyling network protocol has been switched or the object has moved to another peer in the meantime.

Self-Managing Object Connection. Once a remote object has been connected, the method stream controls the underlying network middleware in a self-managing way, e.g. adjusting connection parameters like data compression if necessary. Moreover, if the connection fails for some reason, e.g. due to a network error, the method streams attempts to reconnect the remote object without affecting the business logic. In the same way, relocated objects are transparently reconnected.

Legacy Programming Model. The object binding of network middleware approaches usually requires code customizations, e.g. the use of specific parameter types. Java Remote Method Streaming offers a common approach to bind Java objects independent of binding aspects. Neither extra implementation effort on behalf of the business logic nor recompilations of the source code are required. Plain-Old Java Objects (POJO) created by third-party libraries can be easily networked, too.

4.6.1.3 Approach

The basic idea of the approach is the customizable interception of the object binding between caller and callee by introducing a transparent method stream, as shown in Figure 4.57.

A regular object binding is realized by creating or copying an object reference that is inseparable bound to the object and can be used to pass method calls from the caller to the callee. Once an object reference is deployed, there is no way to change the object communication between caller and callee, e.g. replacing an object pointer with a dynamic network stub. From this point of view, a method stream represents an object reference that is bound to the callee but still allows customizing the object communication. To this end, a method stream separates the object binding from the object communication, as shown in Figure 4.57. Each method call issued by the caller is serialized by the dynamic connector, passed down the method stream and deserialized by the dynamic broker for calling the related method on the callee side.

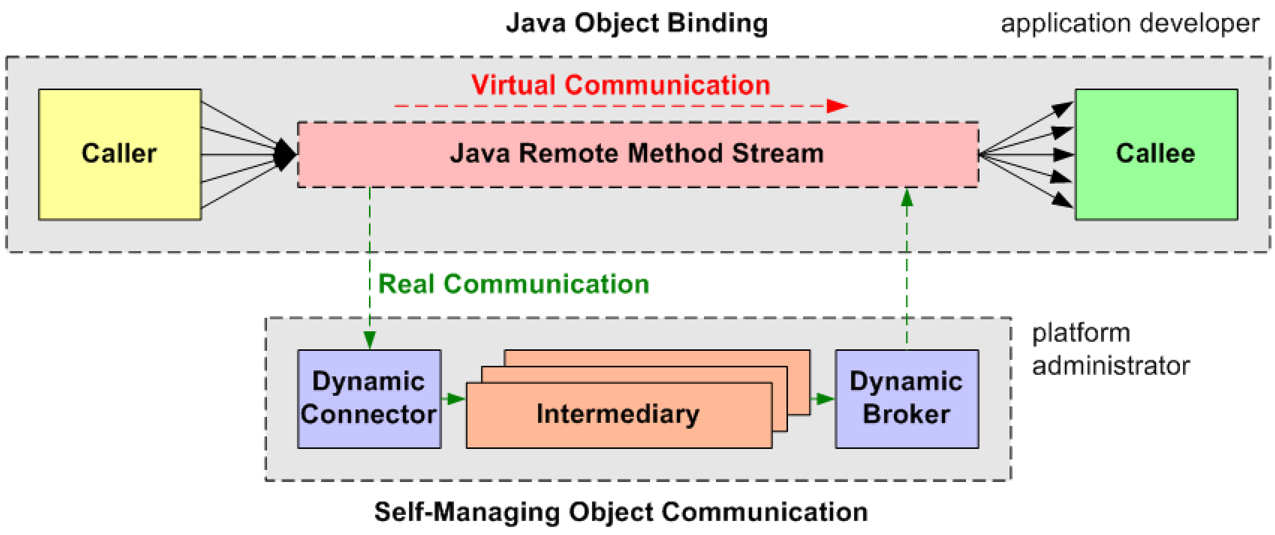


Figure 4.57: Self-Managing Object Communication

While being streamed, the method call can be intercepted and customized by *intermediaries* in a uniform way [25], e.g. to encrypt and decrypt the method parameters or to use a different network middleware for reaching a remote object. In this context, intermediaries can be used to implement a self-managing object communication for bridging platform boundaries in a changing and heterogeneous network scenario. For example, an intermediary can reestablish a broken network link without application intervention. Furthermore, a method stream mimics a legacy object reference, and can connect caller and callee without the need to modify existing code. Hence, it is particularly suitable for networking existing object implementations, e.g. for seamlessly wrapping a legacy library with a custom network interface. Finally, it should be stressed that method streams are not supposed to substitute matured object middleware approaches such as CORBA or RMI. In contrast, method streams try to utilize and combine installed features of the currently employed platform to hide heterogeneous implementations from the application and negotiate an appropriate communication link in the background.

4.6.1.4 Realization

Based on the separation of application-specific object bindings and platform-related object communication, the realization of remote method streams introduces a binding manager for managing object bindings and a connection manager for managing object connections, as shown in Figure 4.58.

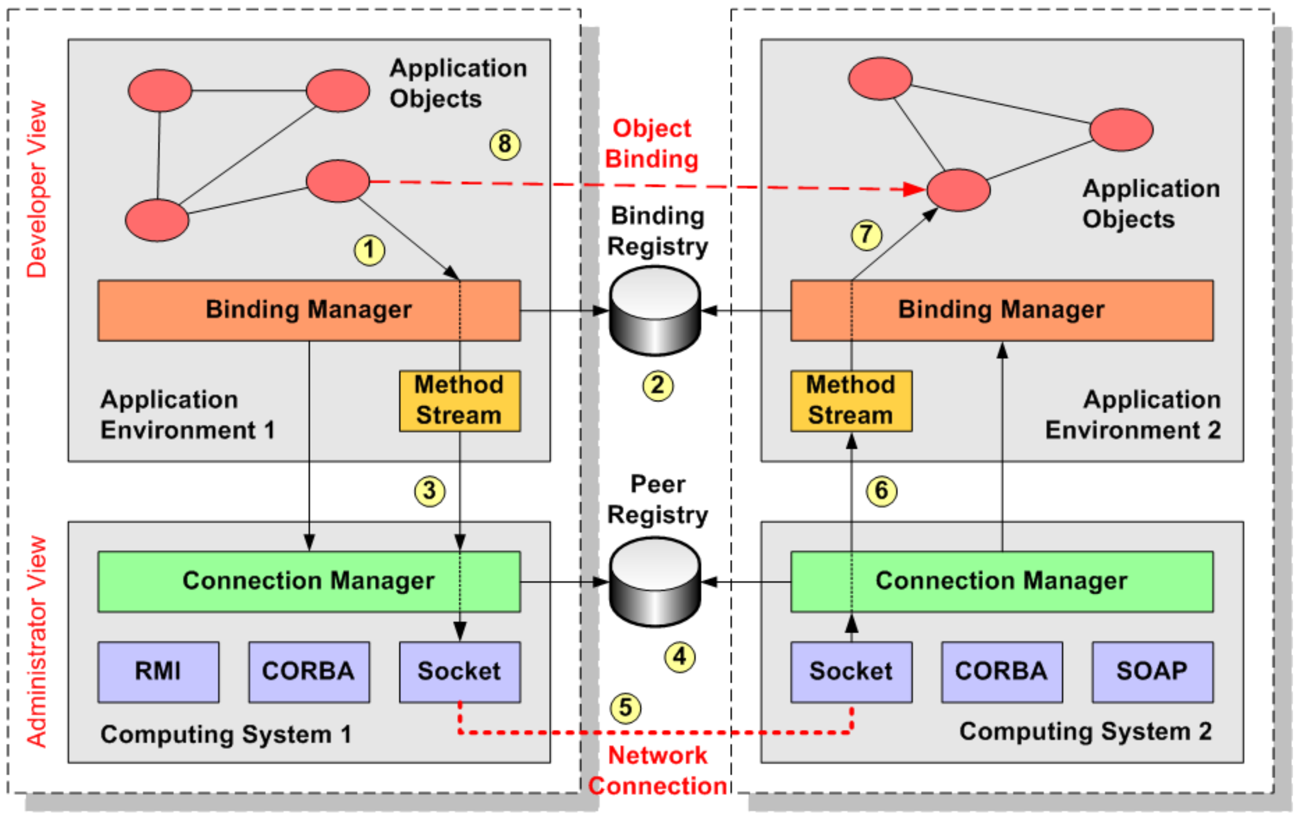


Figure 4.58: Cross-Platform Object Interconnection using Java Remote Method Streams

A peer registry is introduced for registering peers and their connection capabilities, and a binding registry for registering objects and their hosting peer. In practice, a common directory service approach such as LDAP is used that is not bound to a certain object middleware approach like the RMI registry but can be used independently. When a computing system is started, the peer registers itself to the peer registry and announces the communication protocols available for connecting objects. Later on, application objects that want to be reachable by other objects, possibly located on different peers, are registered with the binding registry using a unique binding id and the peer id. Actually, the binding registry does not contain any information about how to communicate with an object and where the object actually resides. The concrete communication path is dynamically resolved and negotiated using the peer registry and evaluating the communication protocol implemented by caller and callee. In detail, if an object binding should be established, the requesting application object (caller) asks the local binding manager for creating an object binding (1) by providing the unique binding id. The binding registry is queried for the object (2) and the information about the hosting peer is passed to the connection manager (3). Then, the peer registry is used to resolve the location of the peer (4) and protocol parameters for establishing a connection, e.g. a network connection via a TCP socket link (5). The remote connection manager creates a method stream (6) that is bound to the remote application object (callee) by the binding manager (7). After all, the caller gets a transparent method stream linked to the callee without noticing the communication path. In this context, method parameters that are serializable will be directly transmitted. However, object references are seamlessly replaced by the binding id of the referenced object and on the remote site replaced with a method stream linked to the original object, as detailed below.

Method Stream Components. The basic components of remote method streaming are *Java Dynamic Proxies* as well as *Java Reflection Dispatcher* [339], as shown in Figure 4.59.

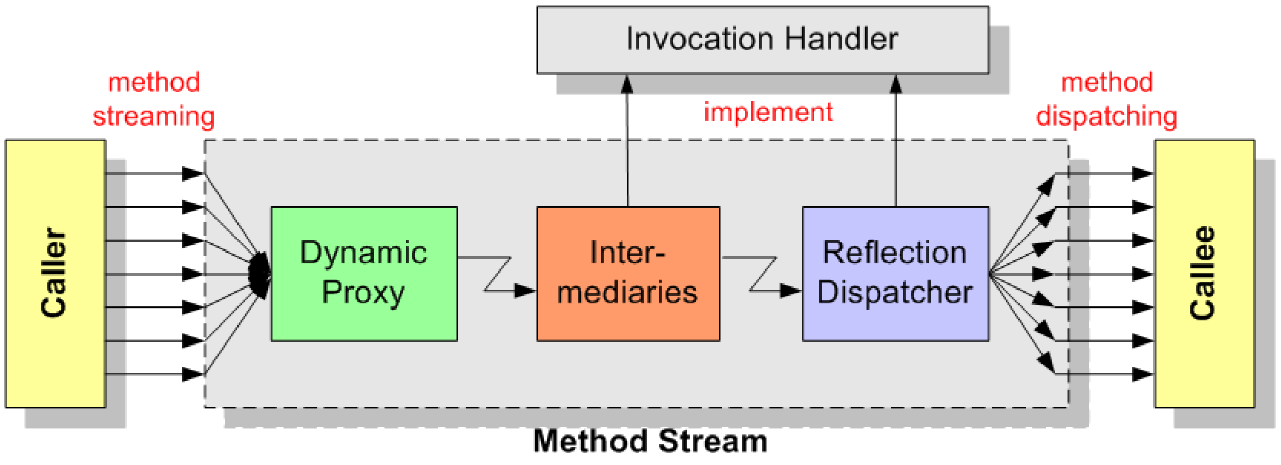


Figure 4.59: Remote Method Stream

Dynamic Proxy. The Java dynamic proxy represents the head of the method stream and is able to masquerade itself with the interfaces of the actual object; hence it takes up the role of the callee still residing on the other side of the method stream. This way, the caller is not aware of the method stream and passes all method calls to the proxy instead. Next, the proxy converts the method calls into a streamable representation and passes them down the method stream to the next link, as shown in Figure 4.60.

The actual method call is encapsulated in a serializable object of type `CMethodRequest` which passed to the next link in the stream. After the method has been processed, a response object of type `CMethodResponse` is passed back through the method stream. If the response is an instance of `CMethodException`, an exception occurred during the method execution and the related exception is thrown to the caller, otherwise the regular return value is returned. The rest of wrapping and unwrapping the method call and its return value is done by the Java Dynamic Proxy mechanism behind the scenes and is explained in detail in [271].

```
public class CMethodStreamHead implements InvocationHandler
{
  public Object invoke(Object obj, Method m, Object[] params) throws Throwable
  {
    // getting the parameter types of the method call
    String[] arr = new String[args.length];
    for (int i=0;i<args.length;i++)
    {
      arr[i] = m.getParameterTypes()[i].getName();
    }

    // constructing the request object
    IRequest req = new CMethodRequest(m.getName(), arr, params);

    // invoking the method
    IResponse resp = link.invoke(req);
    if (resp instanceof CMethodException)
    {
      throw ((CMethodException)resp).getCause();
    }

    return resp.getReturnObject();
  }
}
```

Figure 4.60: Method Stream Proxy

Intermediaries Chain. In addition to stream the method call from one object to the other, the method call can also be inspected and easily modified the way down the method stream using particular intermediaries [25]. They are linked into the method stream and can transparently add particular features such as encryption or logging capabilities. To some extent, they can also be employed to implement cross-cutting aspects like access control, as shown in Figure 4.61.

```
public class CAccessControlIntermediary extends AIntermediary
{
  public CMethodResponse invoke(CMethodRequest req)
  {
    if (accessController.checkAccess(req))
    {
      return super.invoke(req);
    }
    else
    {
      throw AccessControlException(req);
    }
  }
}
```

Figure 4.61: Access Control Intermediary

In this context, a particular task of intermediaries is the creation of cross-platform method streams when caller and callee do not reside on the same platform. In this case, the method stream is actually divided in two parts that transparently communicate with each other using an appropriate real communication link. To this end, the first part on the caller side ends with a particular connector intermediary depending on the real communication link to be used, as shown in Figure 4.62.

```
public CMethodResponse invoke(CMethodRequest req) throws Throwable
{
  // creating output/input streams bound to a socket connection
  ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
  ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
  try
  {
    // sending request
    oos.writeObject(req);

    // receiving request
    CMethodResponse resp = (CMethodResponse) ois.readObject();
  }
  catch (Exception x)
  {
    return new CMethodException(x);
  }
  finally
  {
    oos.close();
    ois.close();
  }
}
```

Figure 4.62: Excerpt of a Connector Intermediary

In this simplified example, a raw TCP socket is used to transmit the method call over the network. The input and output stream are used to write the method request and to read the method response that is returned to the caller at the end. In the case an exception has occurred, the exception object is wrapped within a method exception object and returned instead. On the callee side, a related broker intermediary handles incoming communication requests and passes the method calls down the second part of the method stream on the callee side, as shown in Figure 4.63.

Similar to the connector intermediary, the broker uses the input and output streams of the socket to read and write the method request and response, respectively. Thus, the actual transmission of the method call is separated from the real network communication. Once a network link has been established, the same connection may be shared across distinct object bindings.

```java
public void handleMessage()
{
  ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
  ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());

  try
  {
    // reading request from network
    CMethodRequest req = (CMethodRequest) ois.readObject();

    // calling method on callee
    Object returnObject = link.invoke(req);

    // writing response to network
    oos.writeObject(returnObject);
  }
  catch (Exception x)
  {
    oos.writeObject(new CMethodException(x));
  }
  finally
  {
    oos.close();
    ois.close();
  }
}
```

Figure 4.63: Excerpt of a Broker Intermediary

For full-duplex network links like raw TCP connections, an inbound method stream may be also used to enable outbound method calls. As an example, a task processor located behind a firewall may register to a public task dispatcher and open a method stream to it. If new task processing requests arrive, the inbound method stream is used to transmit the method call from the task dispatcher to the task processor though it is actually located behind the firewall. This is performed in a transparent way without application intervention and can be used to enable public computing with Intranet computing systems as in the ODIX application federation [see Section 5.5].

Reflection Dispatcher. At the end of the method stream, the reflection dispatcher takes the streamed method call and calls the related method on the actual object using Java reflection, as shown in Figure 4.64.

When the dispatcher is created, the actual callee object is passed in the constructor and stored as an attribute `m_callee`. This is later on used to call the passed method with the given parameters. After that, a new `CMethodResponse` object is created and passed back. In case an exception occurred, a `CMethodException` is returned.

```
public class CDispatcher
{
  private Object m_callee;

  public CDispatcher(Object callee)
  {
    m_callee = callee;
  }

  public CMethodResponse invoke(CMethodRequest req)
  {
    // getting the class loader of the surrogate object
    ClassLoader loader = m_callee.getClass().getClassLoader();

    // loading the classes for retrieving the method
    int nArgs = req.getParameterTypes().length;
    Class[] arrCls = new Class[nArgs];
    for (int i = 0; i < nArgs; i++)
    {
      String tp = req.getParameterTypes()[i];
      Class cl = loader.loadClass(tp);
      arrCls[i] = cl;
    }

    try
    {
      // retrieving the method to be called
      Method m = loader.getMethod(req.getMethodName(), arrCls);

      // invoking the method and returning the response object
      Object returnObject = m.invoke(m_callee, req.getParams());

      return new CMethodResponse(returnObject);
    }
    catch (InvocationTargetException e)
    {
      return new CMethodException(e);
    }
  }
}
```

Figure 4.64: Method Stream Dispatcher

Object Binding. The use of method streams as well as of the binding registry is transparent to the application except for binding and connecting the initial reference of the first object, respectively. Besides associating an object with a unique binding id, an object can also be bound with a well-known binding name to ease the lookup of this object, as shown for the binding name `loginservice` in Figure 4.65.

```
IBindingManager bindMgr = CBindingManager.getManager();

IService loginSvc = new CLoginService();

CBindingId bindId = bindMgr.bind("loginservice",loginSvc);
```

Figure 4.65: Binding and Registering a Method Stream with a Binding Name

In an administered and small-scale network environment, e.g. an Intranet, dedicated binding and peer registry servers are installed and announced to the computing systems, e.g. as part of the application configuration during startup. In a large-scale network environment, the Java Remote Method Streaming implementation can switch to a P2P network organizing approach without modifying the application logic. Actually, a *super-peer overlay network* following the *SG-1 approach* has been implemented [245] where super-peers run distributed peer and binding registry services in a self-managing way [247].

Object Connection. On the caller side, the binding id associated with the callee is discovered using the binding registry and the well-known name `loginservice`, as shown in Figure 4.66. After that, the binding id can be used to get the head of the method stream that can be directly casted down to the supposed interface.

```
CBindingId bindId = bindMgr.lookup("loginservice");

ILoginService loginSvc = (ILoginService) bindMgr.connect(bindId);

loginSvc.login(szUser, szPassword);
```

Figure 4.66: Locating and Connecting a Method Stream Using a Binding Name

Consequently, on the one hand the caller gets always a real Java object reference to the head of a method stream that masquerades itself to implement the interfaces of the callee. On the other hand, the method stream is bound to the callee using a binding id and can be dynamically customized with particular intermediaries.

Method Routing. A particular extension of the XDK implementation is the provision of *method stream router services* [see Section 5.5.3] in the ODIX application framework. In a fragmented network environment, they can be used to bridge different network segments, e.g. to enable method calls between object instances located in distinct firewalled Intranets. Similar to traditional network routers, a router service is installed in every Intranet and connected with a public router service. If a method stream is requested to a remote object that is not reachable via a direct network link, the binding registry contacts the local router service and requests a *bridged method stream*. A route lookup is started and if the public router service has an established method stream to the router service of the remote object behind the firewall, that method stream is reused to create an object binding and virtually transmit the method calls across both firewalls. Once the bridged method stream is established, it may be also used for transmitting callbacks from the callee to the caller.

Performance Comparison. Since the realization of *Java Remote Method Streams* is based on *Java Dynamic Proxies* and *Java Reflection*, it of course introduces some runtime overhead in contrast to native Java method calls. However, the overhead depends heavily on the used reflection mechanisms and parameters passed to the method. For example, primitive Java parameter types like long or double have to be encapsulated within objects before they can be used in Java Dynamic Proxies, but object references can be passed directly. Consequently, the approach has been evaluated in two different method calling scenarios [271]. In the first one, a method login is called using object references only, and in the second one a method `sqrt` is defined with `long` as parameter type and `double` as return type. The evaluation has been performed by calling each of the methods 100.000 times, at first on a local object using a regular method call, then on a remote object using RMI and finally using SOAP. Subsequently, a comparable evaluation has been performed using the Java Remote Method Stream approach. The results are shown in Figure 4.67 with a logarithmically scaled y-axis, grouped by local calls and remote method calls using RMI and SOAP.

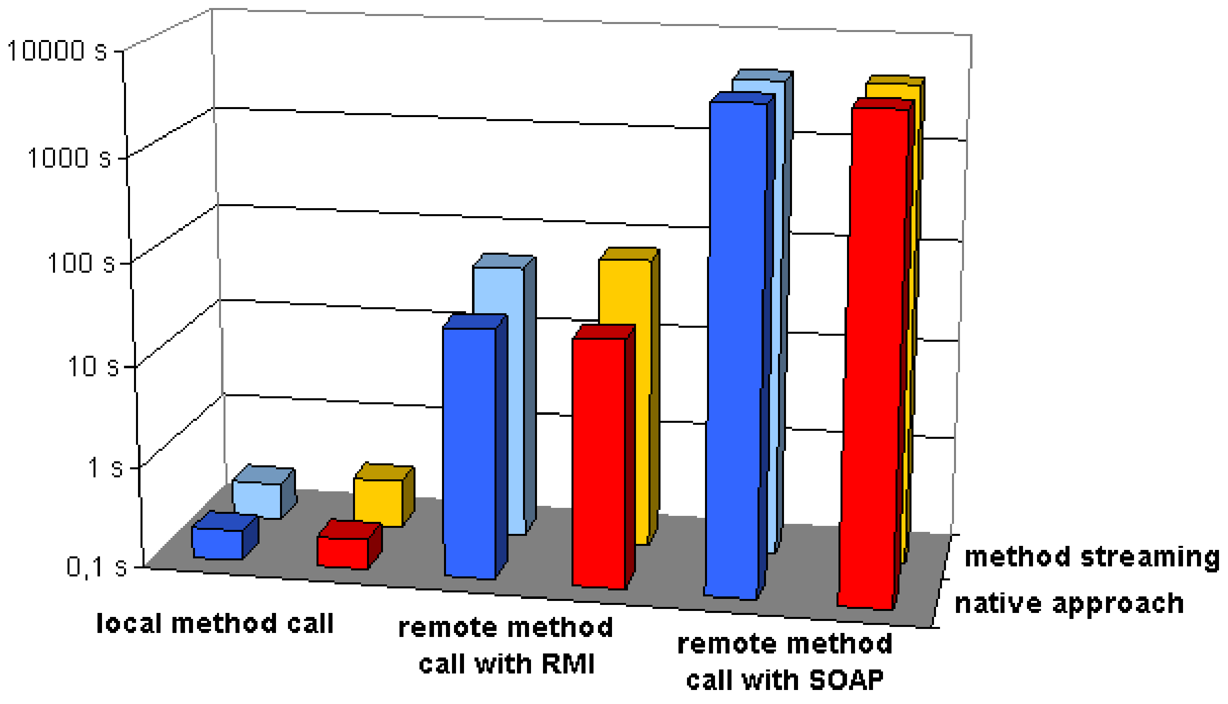


Figure 4.67: Performance Evaluation of Java Remote Method Streams

Each left bar in the group stands for the first method and the right bar symbolizes the second. In addition, the row in front represents the measurements of the native approach and the row in the back the measurements using method streaming. In comparison to the native approaches there is a certain overhead introduced by method streaming, shown as the difference between the first and second row in Figure 4.67. But compared to a local method call, the additional overhead for a remote method call caused by method streaming is relatively small compared to the original overhead as shown in the diagram between the bars in the first and second row of remote method calls and compared further to the bars of the corresponding local method calls. Surprisingly, due to the binary encapsulation of the method call, method streams using SOAP turn to be even slightly faster than the native SOAP approach.

4.6.1.5 Autonomic Application

The Java Methods Stream approach can be used to implement *Autonomic Object Communication* in a cross-platform operating environment with moving remote Java objects and alternating communication protocols, as shown in Figure 4.68.

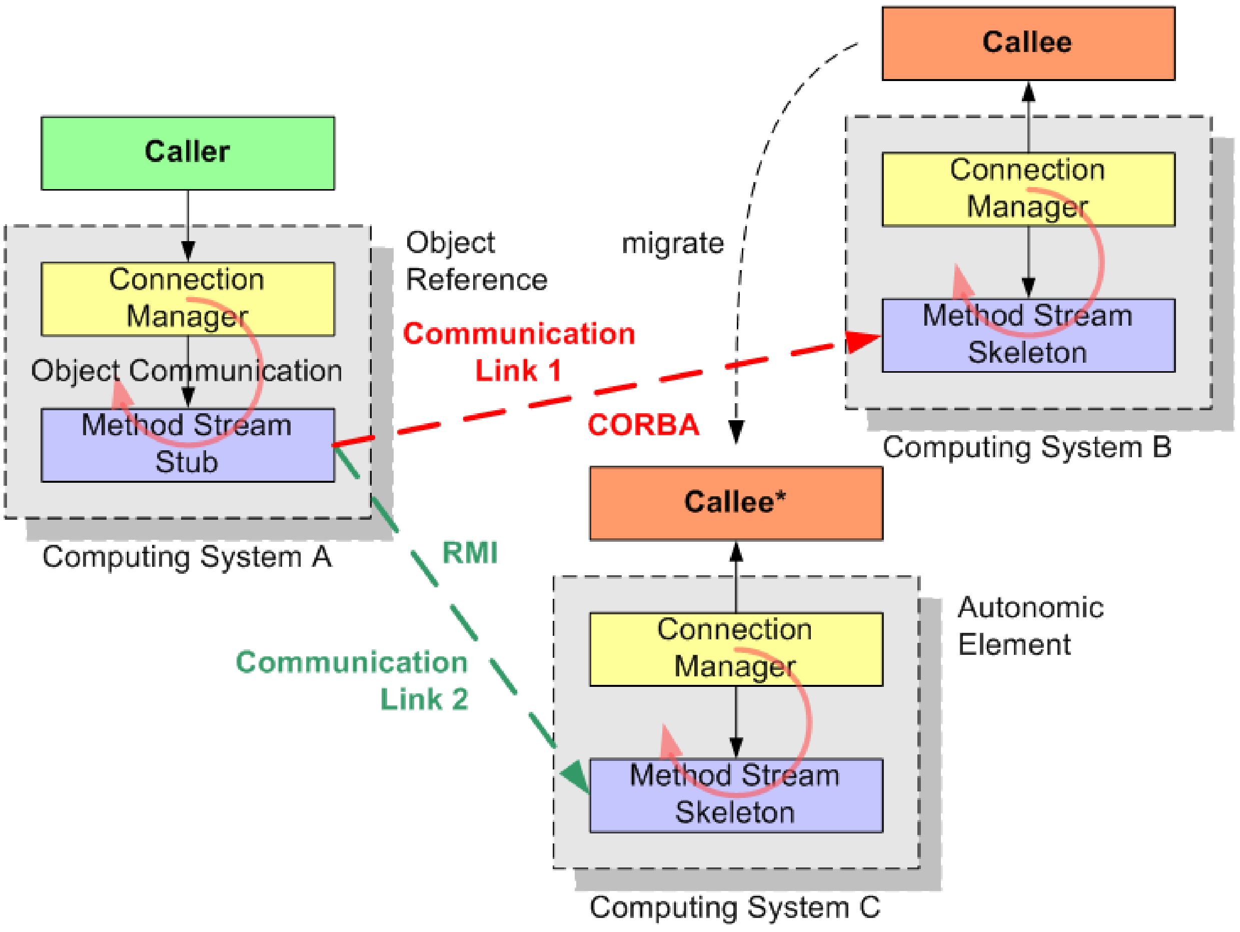


Figure 4.68: Autonomic Object Communication

In a distributed Java computing environment, a caller wants to issue method calls on remote program objects offered by a callee. To this end, an *object link* has to be established between the object reference on the caller side and the object itself on the callee side, e.g. by using an object-oriented middleware approach like CORBA. A particular objective is to perform the method calls on local and remote objects in the same way (location transparency). This is achieved by introducing stubs and skeletons that are created by the connection managers and hide the actual object interaction via the *communication links*.

The autonomic operation is performed by controlling object connection requests and monitoring existing communication links. From this point of view, the callers are the requesting elements, the object references are the autonomic elements, every connection manager is an autonomic manager and the communication links represent the managed elements. In particular, the caller and callee are not aware of handling different communication issues like remote object localization, parameter negotiation or network failures. For example, if a caller wants to connect a remote object, the binding manager is used to determine the hosting peer [see Section 4.6.1.4] and the connection manager negotiates the communication protocol and parameters, e.g. following a connection policy to prefer secure network protocols, if possible *(self-configuration)*. The connection manager usually maps various object links on the same communication link to a peer. This reduces resource utilization and may cause performance decreasing, e.g. since method calls via the same communication link hinders each other. In this case, the communication manager can track the throughput and create additional communication links or switch to another network protocol to improve the performance *(self-optimization)*. Another monitoring task of the communication manager is the detection of broken communication links, e.g. by receiving an exception if a network transmission failed or by periodically pinging the remote peer. Since the object links are loosely coupled with the communication links, the communication manager is able to replace a failing communication link and seamlessly redirect existing object links without user or application intervention *(self-healing)*. Further runtime scenarios have to deal with the modification of communication parameters, the migration of connected remote objects and use of network proxies. The binding and connection managers of related peers communicate with each other and exchange notifications to adjust the communication links in advance before an object link gets invalid. For example, if a connected object migrates to another peer, the origin binding manager sends a notification about the new peer to all connected callers to avoid time-consuming object localization *(self-protection)*.

4.6.1.6 Related Work

There have been many efforts to extend the legacy support of accessing remote objects in Java. For example, some approaches aim at enhancing the existing RMI implementation, e.g. by introducing remote object caching to minimize the amount of transmitted data [92]. Others provide an efficient implementation of the original RMI specification to speed up the network communication, e.g. *Manta* [229]. Concerning the presented features of the Java Method Streaming approach, related work may be separated into proposals that address the *reflective object binding* and proposals that focus on *adaptive object linking*, as discussed below.

Reflective Object Binding. For accessing a remote object, a corresponding object binding in the local process environment has to be established that acts like a genuine object reference. The popular object middleware CORBA offers various ways to configure and customize the object binding aspects. Particular CORBA features are the reflective adjustment of the middleware operation, the handling of dynamic object bindings and their behavioral customization. In *Jonathan*, this is used to extend the Java notion of an object reference and to separate reference management and binding management [91]. For example, the *CORBA interceptor* approach can be used to dynamically add encrypting features when the connection is bridging insecure network sections [252]. In a self-managing scenario, this may be performed by a binding monitor that observes and controls the binding actions and also provides alternatives when an existing network connection fails. Similar to CORBA as a general approach for heterogeneous environments, RMI provides a basis for self-managing middleware in a pure Java environment and offers binding customization by using introspection and remote reflection features [301]. It can utilize special options like *Java Dynamic Proxies* and *Java Reflection* to simplify the development of distributed Java applications, as in *JEDI* [8] and *TRMI* [157], and to transparently customize remote object bindings with smart proxies and interceptors [312]. Advanced approaches like *FlexiBind* introduce pluggable and replaceable policies to support self-managing binding configuration [163]. In *OpenORB*, an extensible binding framework is implemented that is not limited to remote method invocation but also supports different binding types, e.g. event- and multicast-based interactions [290]. Another binding and interceptor approach is supported by the *Java Servlet API* that introduces servlet filters to customize the servlet method execution [171]. Similar to Java Dynamic Proxies, there is a related approach called *Transparent Proxy* [234] in Microsoft .NET. It can be used to transparently bind remote objects by dynamically creating appropriate stubs and interceptors, e.g. to perform security tasks. And though it still lacks features to configure a network communication itself, it could be used as a basis for building a .NET specific autonomic network middleware on top of it.

Adaptive Object Linking. While binding a remote object creates an object reference on the *application level*, linking a remote object establishes a communication path on the *network level*. Object middleware solutions hide the actual network communication from the business logic by using appropriate protocol stubs and skeletons. For example, CORBA and RMI use a standard network protocol like *Internet Inter-ORB Protocol (IIOP)* and *Java Remote Method Protocol (JRMP)*, respectively. While these protocols support the cross-platform operation of CORBA and RMI applications on the network level, it is not possible to switch to another network protocol, e.g. using a HTTP-based communication to penetrate legacy firewalls in the web. A remedy are middleware solutions like *RMIX* that supports the multi-protocol linking of remote objects by using legacy RMI binding, e.g. for tunneling RMI method calls via SOAP [210]. Another approach is *FlexiNet* that introduces a reflective protocols stack and enables the customization of the network communication on different levels by using various client and server side meta objects [166]. In conjunction with the *FlexiBind* approach [163], it represents an advanced Java object binding and linking solution similar to the Java Remote Method Streaming approach. However, they rely on the remote object lookup scheme associated with the currently used network protocol. Thus, there is no strict separation of binding and linking operations, e.g. for looking up the remote object and negotiating the network protocol with the hosting server in a self-managing way before a communication link is established. In contrast to using a dedicated object lookup service in a well-known client-server use case, there are self-managing approaches like *Jini* that use unicast and multicast discovery protocols to obtain a reference to a lookup service in a changing scenario [61, 161], e.g. to support spontaneous networking with distributed services. Jini separates the service announcement and discovery from the actual network communication. The service provider registers a service by submitting a serializable service proxy to the lookup service and the client receives a copy of it when it requests an object reference to the related remote service [189]. This way, the service proxy may use RMI or some other communication protocol in a transparent way to the client application. While Jini supports the dynamic lookup of remote services, it is limited to Intranet scenarios and not suitable for wide-area applications due to its dynamic service lookup approach. In this context, *DACE* provides an abstraction of remote object interaction in a P2P environment by introducing a *borrow/lend* approach and a distributed *publish/subscribe* service [104]. The tight coupling of client and server is broken up and replaced by an asynchronous communication scheme that supports the expiration of remote object references, e.g. if a resource object moves to another peer. Related to large-scale P2P systems, there are object location and data routing approaches, like *JXTA* [140], *Pastry* [306], *Chord* [334] and *Tapestry* [394], that support the building of a self-managing overlay network of peers and can be used to facilitate object linking in a large-scale P2P network.

To summarize, there are many approaches that address the interconnection of distributed Java objects. A common idea is the introduction of an object middleware solution to hide the actual network communication from the application logic. This results in the virtual separation of object binding from object linking and can be utilized to customize each other in an independent manner, e.g. by introducing interceptor and filter objects. Existing middleware approaches and solutions mainly differ in the degree of runtime customization support. For example, *CORBA* is stuck with its default communication protocol *IIOP* while *RMIX* adopts the *RMI* binding approach and can be run with multiple communication protocols like *XML-RPC* and *SOAP*. In turn, *CORBA* offers the customization of method calls by using interceptors while this is not possible with the original *RMI* implementation. Of course, there are proposals using *Java reflection* and *Java dynamic proxies*, e.g. *JEDI* and *TRMI*, which add extra customization options to existing middleware solutions. However, they require more or less specific development of applications to benefit from these extensions, e.g. by adopting the *Jini* implementation guidelines. In this context, reflective and adaptive middleware solutions like *FlexiNet* and *FlexiBind* enable the runtime customization of object and network communication without changing the application logic similar to the presented *Java Remote Method Streaming (JRMS)* approach.

Concerning autonomic operation, various related work can be also used as a foundation to implement self-managing object interconnections to some extent. As an example, *RMIX* can be extended with pluggable transport service providers that may be used by an autonomic manager to monitor and control the actual object communication, e.g. to dynamically switch from one network protocol to another, if needed. A basic advancement of *Java Remote Method Streaming* above related work is the strict separation of object binding and object linking issues. For example, *FlexiNet* encodes the protocol name and object location in the binding name that is used to bind a remote object at the same time. Thus, there is no way to spontaneously switch to another communication protocol without rebinding the remote object. In this context, JRMS introduces an invariant object identifier and supports the self-managing routing of method calls via different network protocols and network boundaries, e.g. in a large-scale cross-platform operating environment.

4.6.2 Java Object Spaces

In a regular Java Virtual Machine (JVM), there is no particular support for managing object interaction among concurrently hosted applications. In this section, Java object spaces are introduced to enable *application-specific object assignment and isolation*. The realization based on the managed separation of object references from object instances is described and its application for task passivation and activation is illustrated.

4.6.2.1 Motivation

A Java application is run by launching a Java Virtual Machine (JVM) and loading the Java class specified on the command-line. By calling the method main, the program execution starts and the application usually begins to create program objects that are interlinked by using object references; hence creating a common object space. As long as at least one object reference exists, the corresponding object is not garbage-collected and remains valid, e.g. the owner of the object reference can call methods and inspect attributes of the object. From this point of view, an object reference operates as an effective substitute for the actual object and there is no way to revoke the object access once an object reference has been deployed. Moreover, object references can only refer to real Java objects and not to other object references; a pointer to pointer approach like in C++ is missing for Java. As a result, every object reference represents an independent instance and can be exclusively updated by its reference holder.

Since the JVM has been designed to host a single application only, it lacks features for creating and managing multiple object spaces, e.g. hosting various applications in a multi-tasking scenario. As a result, there is no way to separate task-related objects within the same JVM or control access on distinct application objects, e.g. private data objects of different user sessions. In well-known multi-tasking application containers, like servlet engines or Enterprise Java Beans (EJB) application servers, the interlinking of application objects is simply avoided. Each application gets a private object space and the programming model requires not to exchange object references directly but to share data by means of the application server. In addition, the regular JVM does not support the replacement of objects without turning existing object references invalid, e.g. for relocating task objects to another host due to a migration request or replacing selected objects after upgrading related code components to a newer release.

4.6.2.2 Features

The overall goals of the *Java Object Spaces* approach are the custom grouping and shielding of task-related objects, and the control of the object communication in a multi-tasking environment. The major features are as follows.

Java Object Isolation. The introduction of Java object spaces allows developers to associate and isolate tightly coupled objects, e.g. all objects created by a Java task, and to ease particular operations like the relocation of the entire object space. Following the Java persistence-by-reachability approach, an object is defined to be a member of an object space if it can be reached by any other object of the same object space via a native Java object reference.

Object Access Control. The objects within the same object group can refer and access each other directly by using native object references. If objects belonging to different object spaces want to communicate, e.g. by calling a method and passing arguments, the communication is checked against given access rules, e.g. allowing access on task-related objects only from object spaces belonging to the same user session.

Custom Object Activation. The regular Java serialization approach handles the streaming of object data only. It does not deal with further actions needed before the object is serialized and after the object has been deserialized. The Java object space approach offers developers to define the passivation and activation procedure of an object space in a customizable manner, e.g. for releasing and acquiring platform resources referenced by related objects, respectively.

Invariant Object References. A native Java object reference is inseparably bound to the referenced object and will prevent the object to be garbage collected; hence an object reference usually never becomes invalid. The Java object space approach allows relocating object groups and their objects though they are still referenced. This is achieved by using *invariant object references* that are updated in a self-managing way.

Hot Component Upgrade. When an object groups is serialized and deserialized, the related class space configuration is stored and restored with the object data, respectively. In addition, the class space configuration can be modified before deserializing the objects, e.g. upgrading the referenced code components to a new release. As long as the object data layout has not been changed compared to the stored class space configuration, the upgrade is performed without particular actions.

4.6.2.3 Approach

The basic idea of the approach is to virtually group application objects into *Java Object Spaces* and to support the common control of inbound and outbound object communication of all encapsulated objects. There are various types of object communication to be addressed in a cross-platform operating environment, as illustrated in Figure 4.69.

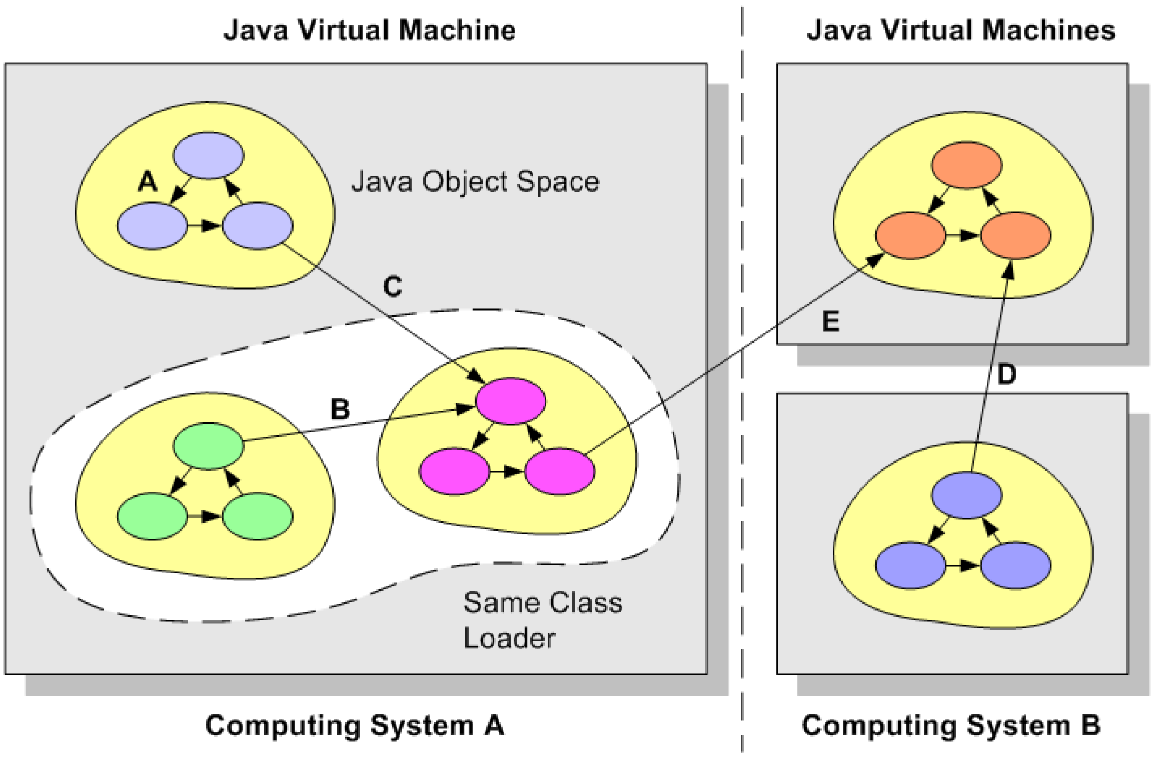


Figure 4.69: Java Object Spaces

There are application objects that use the same class loader, and related object communication will not leave the object space (A). Another scenario is formed by object spaces that belong to different origins but still use the same class loader. Object communication may pass space boundaries, e.g. when accessing objects of an application plugin (B). In the next scenario, object communication involves objects associated with classes that have been loaded by different class loaders, e.g. calling methods on a shared software component (C). This could cause problems with colliding or missing classes depending on the class loader hierarchy and thus requires particular treatment. In case application objects reside in different JVMs, object communication has to cross process boundaries and use object-oriented middleware approaches, such as RMI or CORBA (D). Finally, application objects may be also hosted in JVMs started on different machines. Apart from particular communication issues, such as establishing a suitable network link to the remote host (E), the employed middleware approach has to dynamically resolve the location of an object if objects migrate to another host while they are in use, e.g. by means of an object registry.

4.6.2.4 Realization

The realization is based on using the *Java Method Stream* approach presented in Section 4.6.1 to intercept and check the object communication from and to the contained objects of a Java object space. In addition, an object space manager is introduced to control the operation of various object spaces within the same JVM, as shown in Figure 4.70.

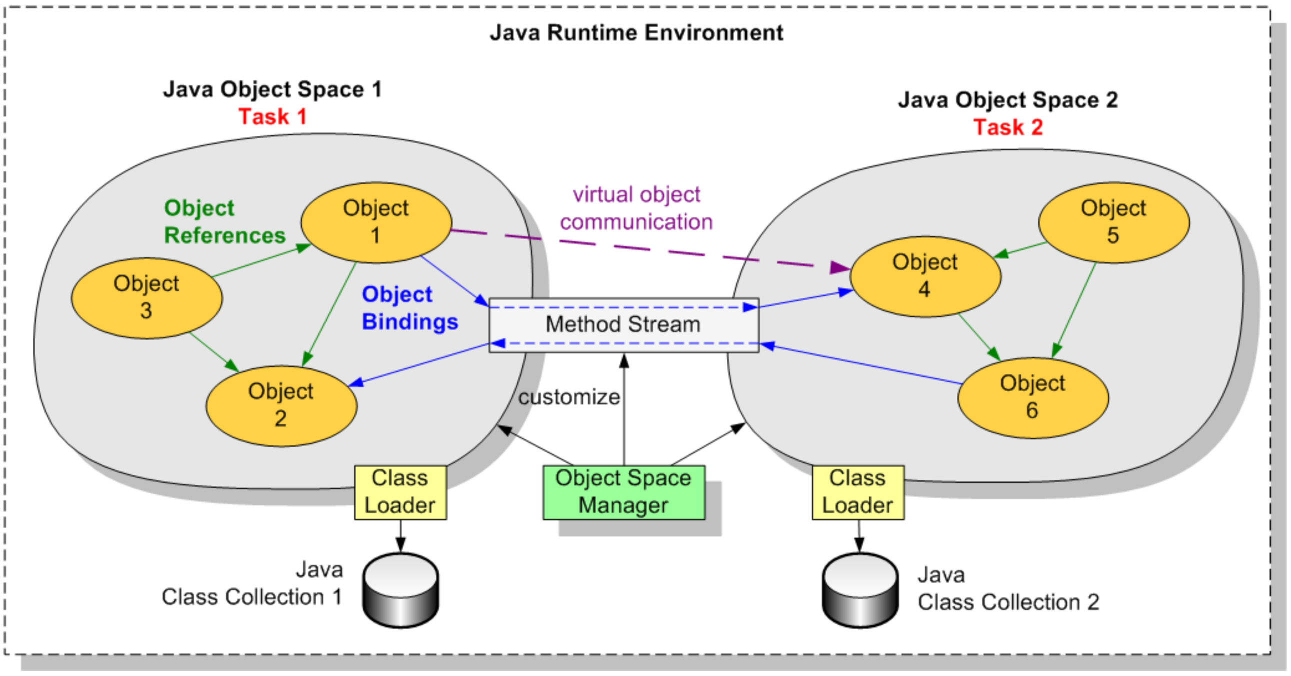


Figure 4.70: Object Communication Control

The objects within the same object space can refer to each other using legacy Java object references and their communication is not affected. Objects from another object space are not directly referenced but bound by a method stream that is dynamically created whenever an object reference passes the boundary of an object space. While method streams can be easily used to connect remote objects in a cross-platform operating environment as described in Section 4.6.1, there is a particular problem when passing method calls among objects instantiated by different class loaders in the same JVM. A method stream can be bound to only one class loader and it is not aware of object spaces using different class loaders. A possible remedy is to serialize every object communication even if the caller and the callee reside in the same object space or use compatible class loaders. However, this would decrease the overall performance and therefore the object space manager customizes the method stream by introducing intermediaries that replace object references only as needed.

Object Adaptation. When a method call leaves an object space and enters another one, it is passed down a method stream and the corresponding request and response objects are transparently serialized and deserialized. At this point, particular binding streams are added that take care of different class loaders of the involved object spaces, as shown in Figure 4.71.

```
public class CBindingOutputStream extends ObjectOutputStream
{
  protected Object replaceObject(Object obj)throws IOException
  {
    if (obj instanceof Serializable)
    {
      // no need to replace with binding;
    }
    else if (obj != null)
    {
      // replace object reference with binding id
      CBindingId bindingId = bdgMgr.bind(space, obj);
      obj = bindingId;
    }

    // return replaced object
    return super.replaceObject(obj);
  }
}
```

Figure 4.71: Binding Output Stream

The legacy Java class `ObjectOutputStream` is used to derive a custom binding output stream that processes the serialized request objects passed from the caller to the callee. In doing so, the overridden method `replaceObject` is called for each object in the stream. While a legacy object output stream would throw an exception if a non-serializable object is found, a non-serializable object is taken out of the stream, a dynamic skeleton is created and bound to the object space of the method stream. Then, the globally unique binding id of the skeleton is put into the output stream in place of the actual object. In effect, all leaving object references to non-serializable objects are encapsulated with dynamic skeletons and the related binding ids are passed to the callee. There, the binding ids are replaced with method streams to the remote objects left on the caller side. The corresponding binding input stream is shown in Figure 4.72.

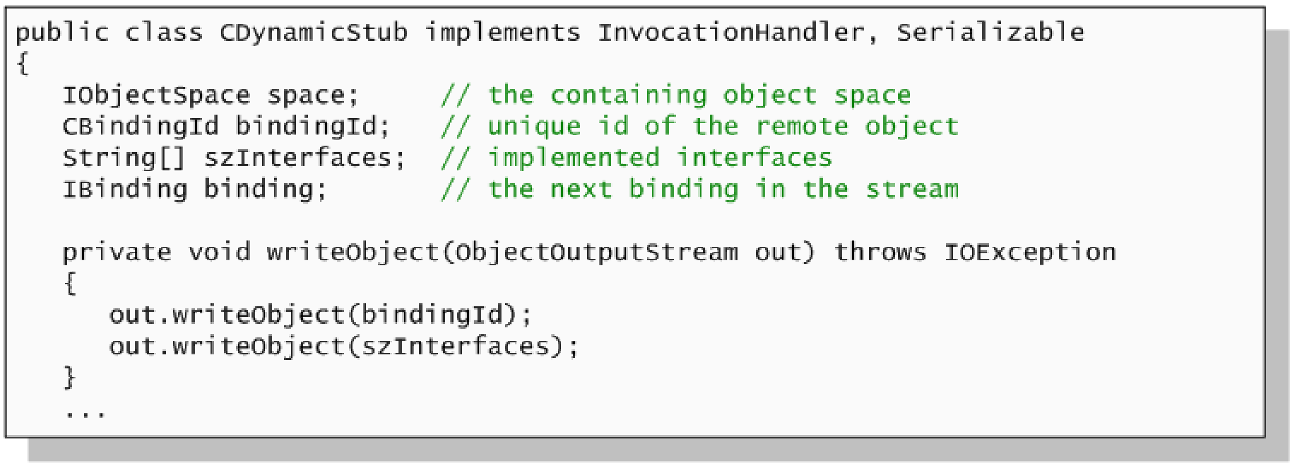
Similar to the binding output stream above, the overridden method `resolveObject` of the binding input stream is called for each deserialized object. The corresponding Java class is evaluated to identify the binding ids that actually represent remote object references. Next, the binding manager is used to connect the related object.

```
public class CBindingInputStream extends ObjectInputStream
{
  protected Object resolveObject(Object obj) throws IOException
  {
    // checking for binding id which indicates a replaced object reference
    if (obj instanceof CBindingId)
    {
      // connecting remote object
      obj = bdgMgr.connect(((CBindingId)obj), space);
    }

    // returning connected object instead of binding id
    return super.resolveObject(obj);
  }

  protected Class resolveClass(ObjectStreamClass desc) throws IOException,
                                                        ClassNotFoundException
  {
    // retrieving the class loader for the object space
    return space.getClassLoader().loadClass(desc.getName());
  }
}
```

Figure 4.72: Binding Input Stream

It should be stressed that the binding manager evaluates the binding id and the object space where the new object reference is to be created. If the target object resides in the same object space, the binding manager returns an object reference that directly refers to the target object, and no method stream is established. In case of different object spaces, a local method stream is used and appropriately wrapped by a Java dynamic proxy. If the object is located in a different JVM or on another host, a remote method stream is established, and the dynamic stub is linked to the method stream. Subsequent method calls will be serialized using the presented binding input and output streams while they are passed through the method streams. In addition, the method `resolveClass` is overridden to modify the class loading required to appropriately deserialize received objects. The class loader is taken from the object space that contains the callee object. As a result, all serialized objects and dynamic stubs of an object space are created by the same class loader. This is important for seamlessly casting object references, updating object implementations and serializing application object spaces, as illustrated below.

Object Space Serialization. A challenging objective of Java object spaces is the serialization and deserialization of an entire object space. The Java Runtime Environment (JRE) already offers an easy-to-use serialization approach that is enabled by the tagging interface `Serializable`. Each object can be decorated with this interface without imposing the implementation of any particular method. It is actually a marker for the Java compiler to create specific methods that are called by the JVM when this object is serialized. Besides serializing primitive attributes, the built-in serialization approach tracks each object reference and automatically serializes the linked objects (persistence-by-reachability). Though an object reference to a Java dynamic proxy can be generally used in place of the regular object reference, the Java serialization approach still differs between a regular object and a Java dynamic proxy object. Instead of transparently passing the serialization calls to the linked object, Java attempts to serialize the dynamic stub. While this approach eases the transparent serialization of regular Java objects, it may cause problems for particular objects, such as dynamic stubs used in the method stream approach. Because of that, Java allows to add custom methods `writeObject` and `readObject` to the object implementation that are seamlessly called instead of the compiler-generated serialization methods. This particular feature is exploited to customize the serialization of the stub by introducing a custom method `writeObject`, as shown in Figure 4.73.



```
public class CDynamicStub implements InvocationHandler, Serializable
{
   IObjectSpace space;      // the containing object space
   CBindingId bindingId;    // unique id of the remote object
   String[] szInterfaces;   // implemented interfaces
   IBinding binding;        // the next binding in the stream

   private void writeObject(ObjectOutputStream out) throws IOException
   {
      out.writeObject(bindingId);
      out.writeObject(szInterfaces);
   }
   ...
```

Figure 4.73: Serializing a Dynamic Stub

The method `writeObject` writes the unique binding id and the names of the remote interfaces to the stream. They are used to recreate and relink the stub to the method stream. For this purpose, a method `readObject` is introduced that is called whenever a dynamic stub is deserialized, as shown in Figure 4.74.

A custom object input stream `IObjectSpaceStream` is used to read in the serialized objects. The basic reason is the association with the target object space wherein the objects and stubs should be deserialized. On the one hand, all regular objects are reinstantiated using the class loader of the object space and the standard Java deserialization approach. On the other hand, the custom method `readObject` is invoked whenever a dynamic stub is about to be deserialized. There, the binding id and the interfaces of the remote object are read from the stream.

```
private void readObject(ObjectInputStream in) throws IOException,
                                                     ClassNotFoundException
{
  // setting the binding id and the implemented interfaces of the callee
  bindingId = (CBindingId) in.readObject();
  szInterfaces = (String[]) in.readObject();

  // connecting this stub with callee
  space = ((IObjectSpaceStream)in).getObjectSpace();
  binding = bdgMgr.connect(bindingId, space, szInterfaces);
}
```

Figure 4.74: Deserializing a Dynamic Stub

Next, the stub is connected to the remote object by calling `connect` on the binding manager. In addition to the binding id and the names of the remote interfaces, a reference to the target object space is passed whose class loader should be used to create the Java dynamic proxy. This ensures that the returned object references can be seamlessly casted and passed among objects within the object space wherein this stub resides.

4.6.2.5 Autonomic Application

The Java Object Space approach can be used to implement *Autonomic Object Linking* in a multi-tasking Java runtime environment with the ability to monitor and control the interaction between Java objects of distinct object groups (application partitions), as shown in Figure 4.75.

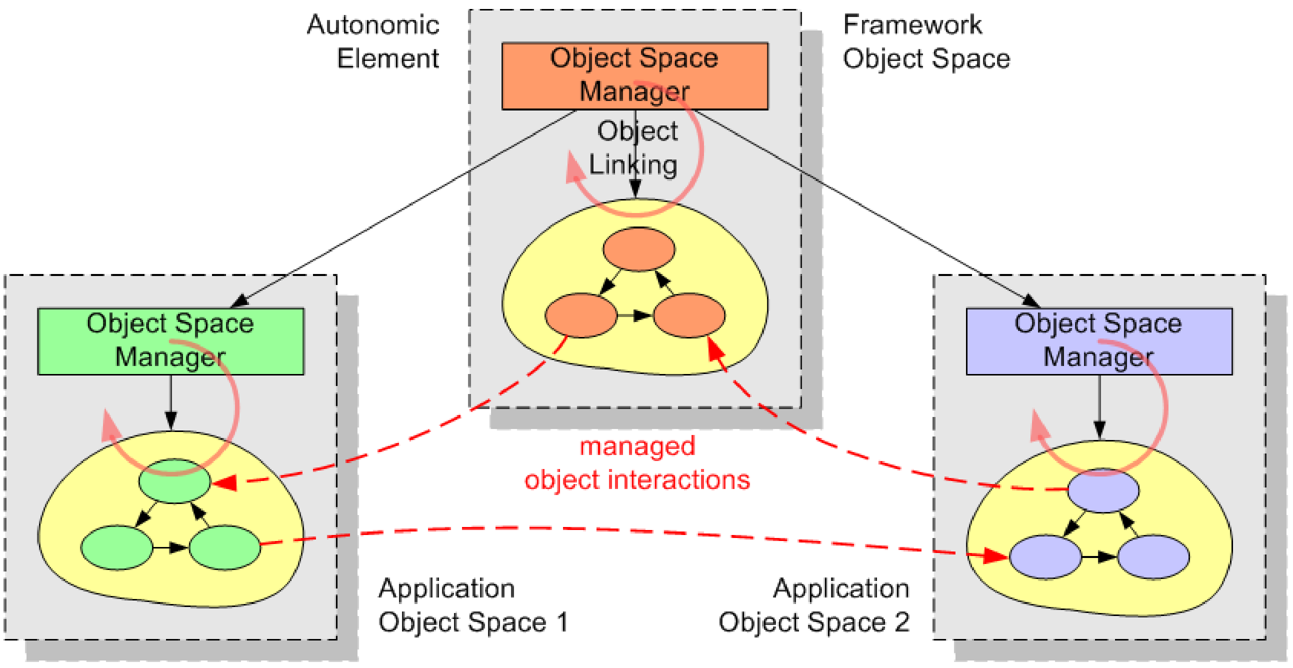


Figure 4.75: Autonomic Object Linking

To initialize the multi-tasking runtime environment, first an application framework is launched and associated with an object space in which all Java objects are automatically grouped. Later on, for every launched application a new child object space is created to host the related application objects. Whenever an object reference passes an object space boundary, e.g. usually as a parameter of a method call or as an attribute of a transmitted object, the involved object space managers replace the original object reference with a particular object link to intercept the object interaction. The autonomic operation is performed by monitoring and controlling the inbound and outbound object communication using the dynamically inserted object links. From this point of view, every object passing an object reference to an outbound object is a requesting element, the object space managers are the autonomic managers, the object links act as the managed elements and every object space is an autonomic element from the perspectives of outbound objects. In particular, the involved objects are not aware of the autonomic link control. For example, if an object link should be established between objects of different application object spaces, e.g. for passing process data to the next node in a workflow chain, the affected object space managers determine if they share a common class loader. If not, the object link has to serialize and deserialize passed objects to decouple incompatible class references of each other's object space *(self-configuration)*. Another option is to serialize the entire object space into a file and deserialize the stored objects in a separate Java process along with the restoration of the execution state [see Section 4.7], e.g. to enable load-balancing in a cluster environment *(self-optimization)*. Similar to this option, the serialized objects may be reloaded in a new object space of the same Java runtime environment while restoring existing object links, e.g. for seamlessly replacing the classes of active Java objects with a security update *(self-healing)*. Finally, the object space manager can monitor and control access on private object spaces, e.g. application object spaces of different users in a multi-session scenario *(self-protection)*.

4.6.2.6 Related Work

The overall goal of *Java Object Spaces* is the separation of an object reference from the related object instance to support custom object grouping. This leads to the custom handling of object communication which has already been addressed by many Java object middleware approaches. In the following review, related work is considered with respect to the objectives and challenges of Java application object isolation in a cross-platform operating environment.

Custom Java Class Loader. There are many approaches towards application object isolation that are based on the introduction of custom class loaders and the modification of how classes are selected, loaded and arranged in the JVM. Popular examples include servlet engines, such as *Apache Tomcat* [171], and J2EE application servers, such as *JBOSS* [113] that benefit from the ability to load different class variants at the same time. Custom class loaders may also be used to update already loaded classes and use new variants, e.g. for reloading and restarting a Java servlet. A common drawback of custom class loaders is their focus on *class separation* and lack of features to support object isolation. It is not possible to update the implementation of already linked objects. A custom class loader is not aware which object belongs to a certain application instance, and there is no way to determine object groups of an application, e.g. scheduled for migration.

Java Component Framework. Another approach is the introduction of component frameworks to handle object interactions and related application containers to control the component loading. Popular examples are *EJB* and *OSGI*. They support reloading and updating single components during runtime. Similar to the custom class loader approach, however, it is not possible to group distinct application objects and to handle the object communication between application partitions within the application container. Various solutions towards *transparent runtime evolution* of Java applications address similar issues concerning application partitioning and maintaining established object references. For example, the *Jadabs application container* uses dynamic proxy objects to control inbound and outbound object communications [357]. It can update component groups without shutting down referencing components that are located in other component groups of the same JVM. For distributed systems, a component framework is proposed in [55] to support reconfiguration and migration of component groups. In particular, it implements transparent switching of local and remote object communication depending on the hosting scenario. A basic issue of component frameworks is its focus on managing components and not object groups associated with an application instance. Thus, it cannot be applied with legacy Java classes and related applications have to be implemented following the component development specification.

Custom Java Virtual Machine. Approaches based on a custom Java Virtual Machine, such as the *Multi-Tasking Virtual Machine (MVM)* [75], *NOMADS* [348] and *Camel* [6], may offer particular features, e.g. a multi-tasking runtime environment, code sharing and a mobile application framework. On the other hand, they require additional effort to install the JVM and suffer from the lack of portability in a heterogeneous cross-platform operating environment. In a similar way, code rewriting approaches, such as *J-Orchestra* [356] and *J-Seal2* [36], are able to incorporate extra features like automatic application partioning, advanced resource control and protection domains. Due to byte code modification during compile time, they inherit the problem that this cannot be performed on-demand during runtime, e.g. when passing object references among uncertain and dynamically loaded components. Furthermore, they rely on a different programming model that hinders the dynamic hosting of unmodified legacy Java applications. In addition, compile-time byte code modifications cannot be applied to signed Java classes without breaking the seal. A particular variant is proposed in [35] that is based on the runtime transformation of byte code to decorate loaded Java classes with application partitioning features.

Java Object Middleware. The next approach considered is legacy *Remote Method Invocation (RMI)*. It is the proposed solution of Sun for binding distributed Java objects and calling methods remotely. There are various approaches that rely on RMI and extend it with custom features [20]. For example, there is basic support to relink RMI object references on the fly by using smart proxies and interceptors [312], although the custom separation of an RMI object reference from its corresponding remote object is formally not supported. In addition, the grouping of tightly coupled objects and the custom interception of object communication from and to this object group is not supported. Another drawback of legacy RMI is the explicit compilation of stubs/skeletons for each object implementation. This is not feasible for on-demand operation. As a remedy, *TRMI* [157] uses Java dynamic proxies to wrap the RMI communication and to enable dynamic remote object linking without using the `rmic` compiler. The same is valid for RMI in Sun J2SE 5. However, the RMI-based approaches do not address multi-application hosting. There is no separate handling of different object instances and custom tracking of object references within the same JVM.

Object Request Broker. Another well-known object middleware approach is the *Common Object Request Broker Architecture (CORBA)*. In contrast to RMI, it readily supports the dynamic linking of yet unknown remote objects and the custom interception of object communication via interceptors [263]. A Java implementation is deployed as part of the Sun J2SE and does not have to be installed manually. From this point of view, CORBA represents a suitable approach to realize Java application object isolation on top of it. On the other hand, the application of CORBA requires some modification of legacy Java application code, such as using the narrowing approach to cast a remote object reference. This may complicate the use in a cross-platform operating environment with legacy application code migrating from one host to another. The update of object implementations leads to a similar problem that CORBA stubs and skeletons have to be exchanged whenever the target object implementation is modified.

In summary, there are various approaches that address and support application object isolation to some extent. Custom Java class loader approaches, e.g. as used in *Apache Tomcat*, focus on class separation and do not support object isolations. Component frameworks, such as *EJB* and *OSGI*, may be used to implement runtime evolution of Java applications but they cannot be used to group custom applications. Custom Java VM approaches, such as the *MVM* and *J-Seal2*, provide particular features but often rely on a proprietary JVM or a different programming model. They cannot be used easily in a cross-platform operating environment with unknown applications. Standard object middleware approaches, such as CORBA, can be used to dynamically link remote object instances and are able to separate the tight coupling of object reference and object instance. Common object middleware, however, typically fail for tracking the object references spread in distinct application partitions within the same JVM.

4.7 Ad Hoc Execution Migration

In a distributed computing environment, programs are typically run in separated execution environments, e.g. web services in different servlet containers of an application server and user applications on distinct desktop computers. Though they are advanced approaches like *Message Passing Interface (MPI)* [128] to synchronize the parallel execution of tasks on separate nodes, each node is still running a prepared application portion that has been deployed and launched in advance. In the on-demand computing approach, computing resources, services and applications are made available to the user as needed without prior deployment and configuration. Extending this idea to running tasks in a distributed environment, an emerging feature is *ad-hoc execution migration* and the seamless movement of task execution from one computing node to another. This approach raises new challenges with respect to *code mobility* as discussed in [126]. In particular, the serialization, transfer and deserialization of execution states and the provision of code components are essential facets of a computing environment supporting execution migration and mobile agents [45]. In Section 4.7.1, an approach to an advanced *Java Thread Controller* is presented that supports virtual controlling of Java program execution by introducing a signal-based thread communication system, e.g. for coordinated suspending and resuming of multiple Java threads. In Section 4.7.2, the conceptual approach and implementation of *Java Execution Units* are described that enables the actual task migration by encapsulating the execution state of a Java program along with the required code and data objects [89, 278]. It enables the serialization, the transfer and the deserialization of a running Java program in a self-managing way.

4.7.1 Java Thread Controller

The regular Java Virtual Machine (JVM) has only limited built-in support for managing Java threads. In this section, an advanced Java threading approach on top of regular Java threads is presented that enables fine-grained monitoring and controlling of Java threads. The realization of the approach using a legacy JVM and particular check points is described, and the application for passivation and activation of running tasks is illustrated.

4.7.1.1 Motivation

To process a task in Java, a JVM has to be launched with the main class of the related Java application passed on the command-line. In this scenario, the JVM is run in a separate user process and executes only one Java application at a time; turning the original JRE into a single-tasking environment. Over time, various multi-tasking approaches have emerged that concurrently execute multiple Java applications within a legacy JVM by starting each in a separate thread and thread group as well [22]. For this purpose, an application framework like a servlet engine is started first and the required code packages of the applications are loaded one-by-one using Java dynamic class loading. Then, the actual program entry point of each application is determined and passed to a new thread that starts with the application execution similar to the original single-tasking approach. Further threads are automatically associated in the thread group of the application which eases the overall task handling, e.g. determining whether all threads of the task have ended.

The concurrent execution of Java applications using multi-threading features implies advanced requirements on the thread management of distinct tasks, e.g. controlling their execution and passing runtime events like an upcoming JVM shutdown. Though there are legacy thread control methods like *suspend*, *resume* and *stop*, their use is not recommended due to object locking problems [217]. Their synchronous and pre-emptive character let the addressed threads no chance to respond upon the request in a structured and decoupled way, e.g. closing network channels and releasing allocated resources in the background before suspending the task. There is also no common programming interface to communicate with a thread in an asynchronous way, e.g. by posting events. Furthermore, the controlling application framework is not able to monitor the thread execution state of tasks apart from being stopped or still running. Thus, advanced multi-thread operations and synchronization are not originally supported, e.g. for implementing task migration and hibernation.

4.7.1.2 Features

The *Java Thread Controller* approach pursues to improve the monitoring and controlling of regular Java threads by introducing particular thread wrapper classes and check points for customizable interception of the thread execution. The major features are as follows.

Java Thread Controller. To control the execution of Java applications in multi-tasking JVM, a process manager is introduced that is able to track and monitor the threads of all tasks. This is valid for registered threads but also for non-registered threads, e.g. daemon threads started by an application using regular thread classes. If a task should be stopped or a login session ends, the process manager determines the related threads and terminates all of them automatically.

Custom Thread State Model. In addition to the legacy Java thread states *new*, *running*, *blocked*, and *dead*, a developer can modify the thread state model and add custom states like *migrating* and *passivated*. This is achieved by introducing a thread wrapper that implements the state model and is dynamically created by the process manager for every registered thread the first time it is controlled, e.g. when suspending all threads of a task and waiting for the related state transition.

Asynchronous Thread Signaling. A particular feature is the ability to control threads in an asynchronous manner. The thread is not directly affected by the caller but it is signalled to change its operation. The signal is represented by a signal object that is posted into a message queue and checked by the thread when passing certain check points, e.g. while calling *yield* or waiting to reach a given thread state.

Synchronous Action Handling. In contrast to asynchronous thread control, the synchronous handling of custom runtime actions allows coordinating multiple threads in a pre-defined way. For example, if the JVM is shutdown, a developer can specify to terminate all application threads at first, then to stop background services and finally to flush data buffers. For this purpose, particular handlers have to be implemented and passed to an action queue that is then processed one-by-one.

Legacy Code Support. The implementation is based on standard Java features and does not force developers to use custom thread objects. In particular, the tracking and monitoring also works for existing threading code, e.g. provided in third-party libraries. However, for proper handling of the thread signals, the developer has to implement a signal handler method or check for incoming signals regularly.

4.7.1.3 Approach

The basic idea of the approach is to separate the *thread control* from the *thread execution* by introducing a signal-based communication, as shown in Figure 4.76.

A Java application does no longer directly interact with a Java thread, e.g. by calling *suspend* or *resume* on the thread object, but uses a *Java Thread Controller* to submit signals into a related signal queue. The addressed thread regularly evaluates the enqueued signals by calling a signal handler. In contrast to regular thread control methods that may interrupt the thread execution anytime, the application developer can decide where to call the signal handler and process waiting signals. As a result, the thread is not accidentally interrupted, blocked or terminated in a critical code section, e.g. without having the chance to clean up resources. Moreover, by using the signalling approach some of the inherent problems of Java thread handling are avoided like lost synchronization monitors when calling *suspend* or *stop* on the thread [340].

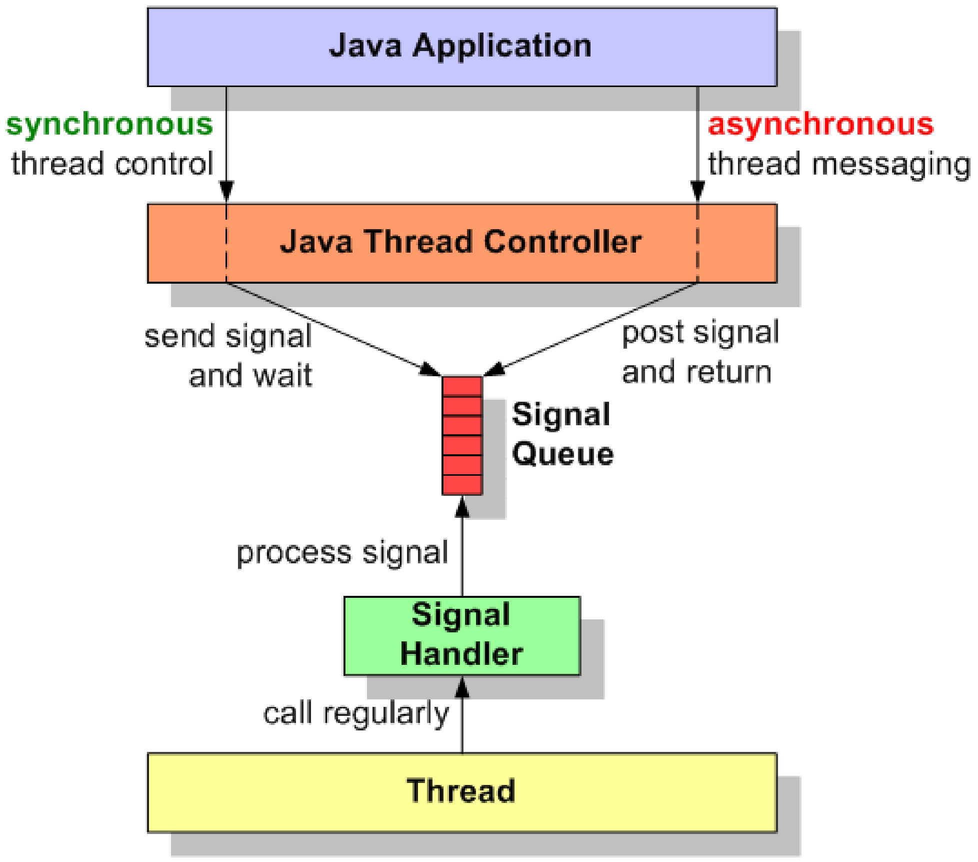


Figure 4.76: Java Thread Controller

Besides common thread control, custom signals allow introducing new thread commands that may be processed by particular thread implementations only, e.g. preparing a thread to migrate its execution to another host. In this context, signals can also be used to exchange specific program data between related threads, e.g. implementing an asynchronous messaging approach in a multi-tasking Java runtime environment. Finally, signals may be decorated with priorities to indicate urgent requests that have to be processed as soon as possible, e.g. signalling an upcoming shutdown of the JVM.

4.7.1.4 Realization

The realization of the approach is based on the introduction of a thread wrapper with a signal queue and the extension of the actual thread implementation with a signal handler, as shown in Figure 4.77.

To control a thread, a global thread manager creates a thread wrapper that gets a reference to the thread object and passes a reference of the thread wrapper back to the thread. A call of a thread control method (1) enqueues a corresponding signal into the signal queue of the thread wrapper (2). The next time the thread calls the method *yield* on the thread wrapper (3), the standard signal handler of the Java Thread Controller is called (4) and evaluates the signals (5) related to common thread controlling operations like *sleep*, *wait*, *suspend* and *resume*. Afterwards, a configurable custom signal handler is called (6, 7), if available, and can be used to process both standard and application-specific signals, e.g. closing a file handle before suspending the thread execution or notifying all application threads about a migration request.

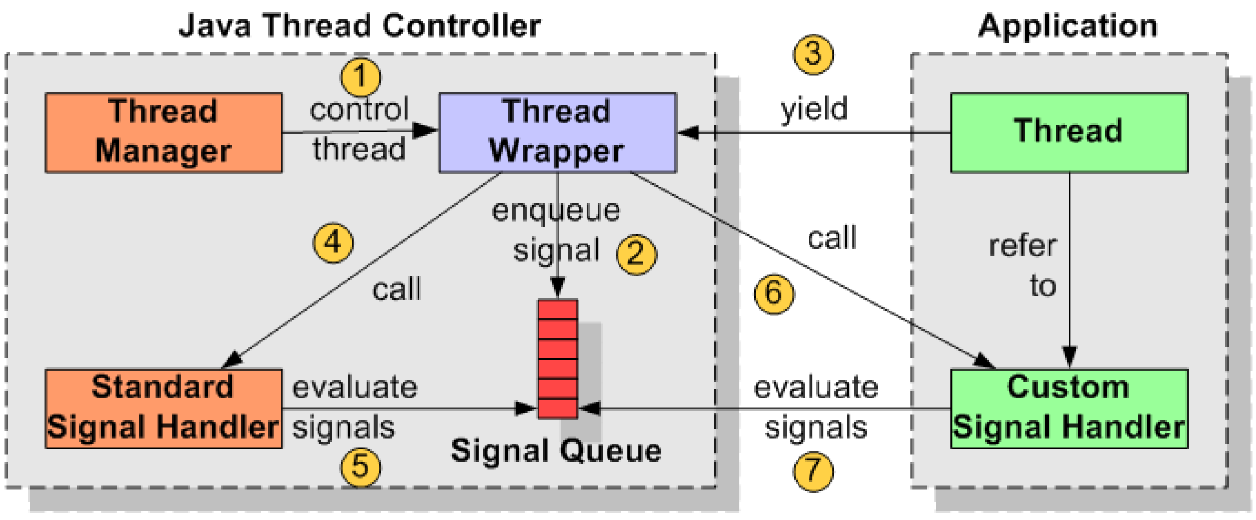


Figure 4.77: Java Thread Wrapping

Managed Thread Control. A particular idea of the approach is to let the thread determine when to handle a controlling request and to potentially interrupt the regular thread execution, as shown in the example for managed thread control in Figure 4.78.

If a thread should be started, the thread controller first creates a thread wrapper object that in turn creates the actual Java thread object. The thread is initialized and configured with the *runnable* object that contains the business logic that should be executed in a new thread. In this context, the thread wrapper evaluates the implementing feature interfaces of the *runnable* object by using Java Reflection and subsequently calls `configure`, `contextualize` and `compose`, similar to a component object [see Section 4.3.2]. Then, the original method `run` is called and the thread control is passed to the *runnable* object. Later on, if the thread manager wants to control the thread, it calls a method `signal` on the thread wrapper object that adds a related control signal to the signal queue of the thread wrapper, and may return immediately or wait for the signal to be processed. Each time the *runnable* object calls the thread wrapper method `yield` at an appropriate *check point* in the execution that may be interrupted, the thread wrapper uses `handleSignal` to let the associated signal handler evaluate waiting signals. Besides processing custom signals, the presented approach is also used to manage regular thread control operations, as shown in Figure 4.78 for suspending and resuming a thread. If the standard signal handler encounters the signal *suspend*, it remains in an endless loop and usually waits for the signal *resume* before the thread execution is continued. However, the thread may still receive different signals, e.g. *shutdown*, and the application developer is free to implement custom signal handlers that let thread react on external events while actually being blocked, e.g. end the thread though still suspended.

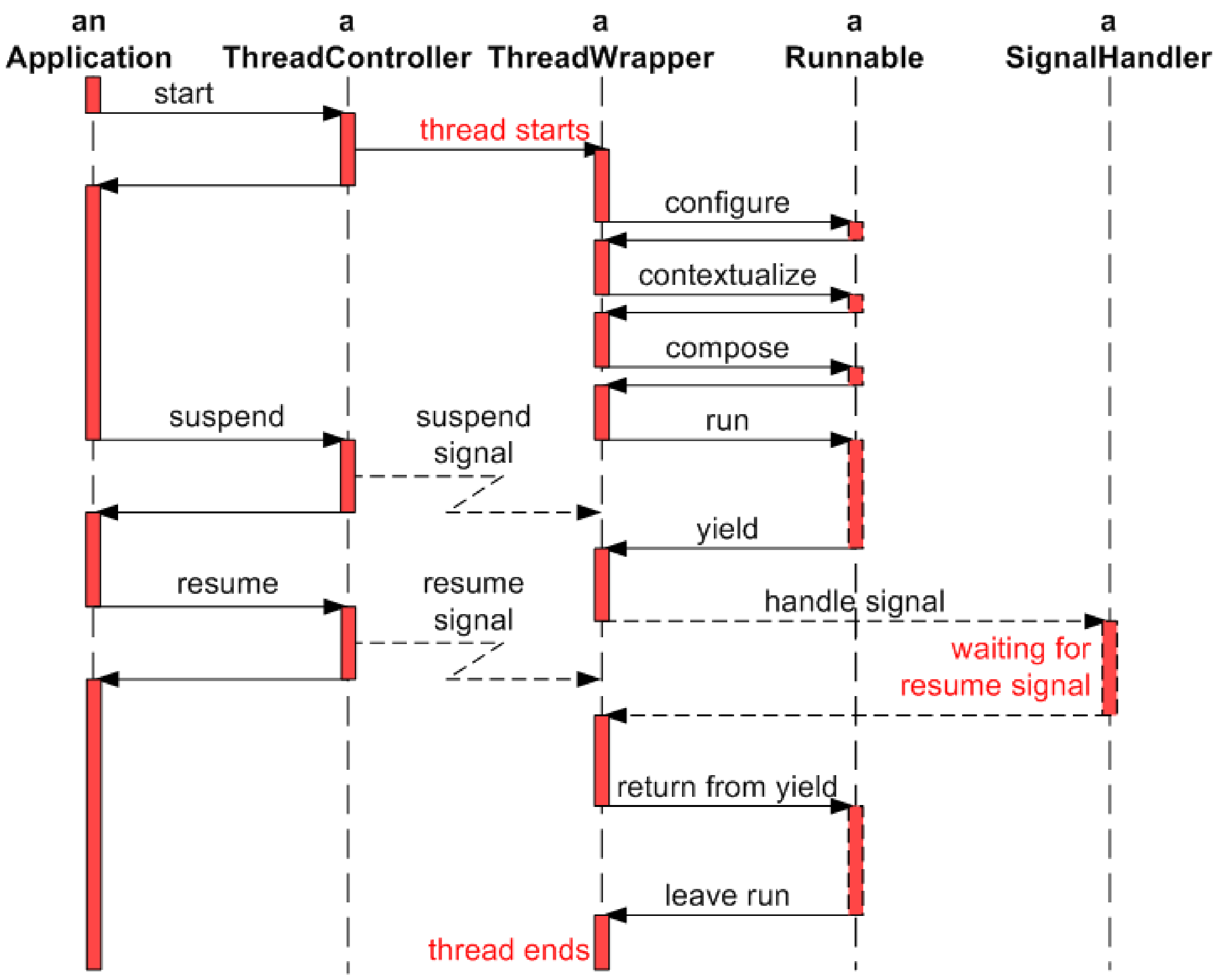
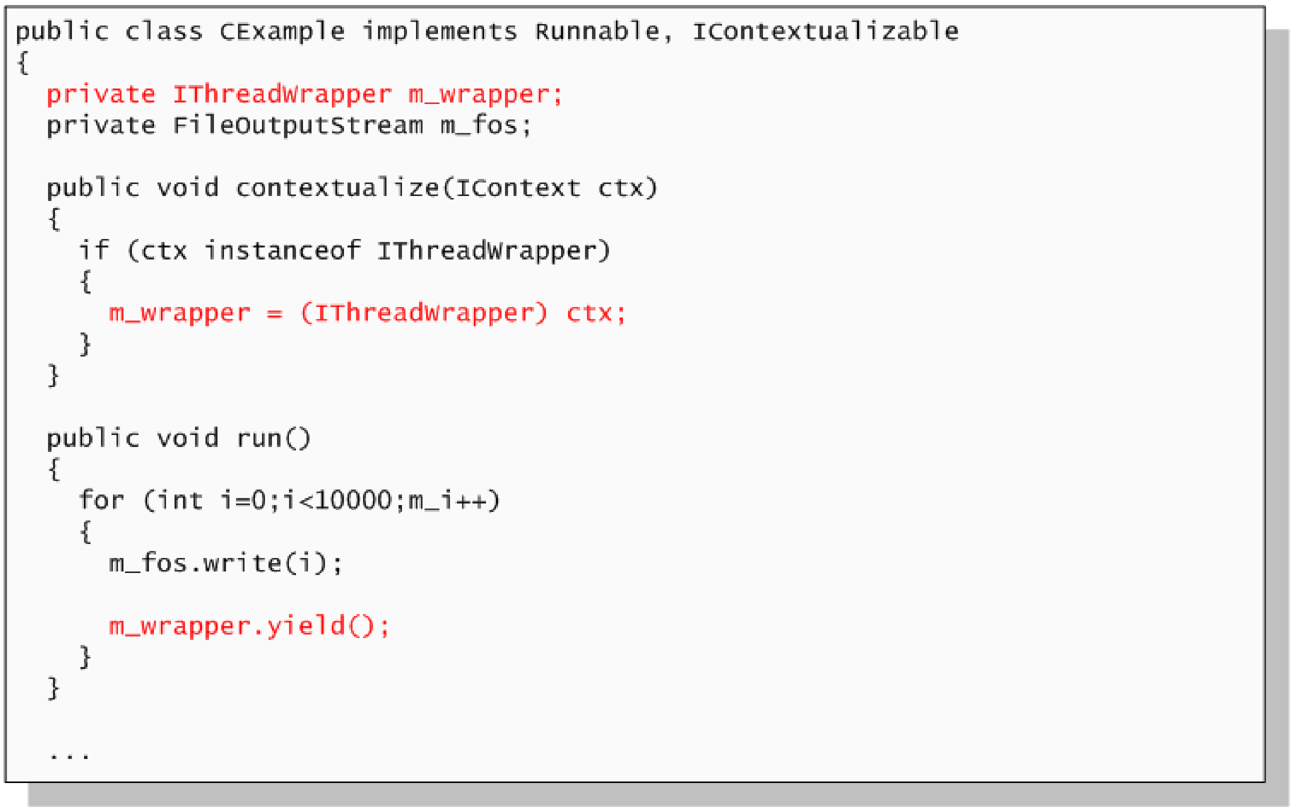


Figure 4.78: Example for Managed Thread Control

Thread Feature Implementation. As described above, a thread implementation can implement specific feature interfaces for being controlled by a Java Thread Controller, as shown in Figure 4.79.

The legacy Java interface `Runnable` indicates that `CExample` contains a method `run` and may be executed in a separate thread. The method `contextualizable` is part of the interface `IContextualizable` and called by the thread controller to pass an object reference to the thread wrapper during the initialization phase. If the thread is executed, the method `run` writes integer values into the `FileOuputStream` `m_fos`. For every iteration, the thread wrapper method `yield` is called that let the thread wrapper check the signal queue and call the standard signal handler. Thus, if an application sends a signal *suspend* to the thread, the execution would be interrupted during the call of `yield`, actually a check point, and blocked until a signal *resume* is received.



```
public class CExample implements Runnable, IContextualizable
{
  private IThreadWrapper m_wrapper;
  private FileOutputStream m_fos;

  public void contextualize(IContext ctx)
  {
    if (ctx instanceof IThreadWrapper)
    {
      m_wrapper = (IThreadWrapper) ctx;
    }
  }

  public void run()
  {
    for (int i=0;i<10000;m_i++)
    {
      m_fos.write(i);

      m_wrapper.yield();
    }
  }

  ...
```

Figure 4.79: Thread Feature Implementation

Custom Signal Handling. For application-specific signal handling, the thread can additionally implement the interface `ISignalHandler`, as shown in Figure 4.80.

The method `handleSignal` of the interface `ISignalHandler` is implemented by `CExample` to receive and evaluate thread signals. When the custom signal *started* is received, the signal handler can initialize thread resources before the method `run` is actually called, e.g. by opening a `FileOutputStream`. Similar to this, the signal handler may release resources when receiving the signal *stopped*. In addition, the custom signal handler can be used to prepare the thread before suspending and after resuming the execution, e.g. releasing and requesting thread resources. By implementing application-specific signal classes, developers can use the signal handling to trigger appropriately implemented threads with custom data, e.g. notifying all threads of a user session about a forthcoming session termination.

```
public class CExample implements Runnable, IContextualizable, ISignalHandler
{
  ...

  public synchronized void handleSignal(ISignal signal) throws XException
  {
    if (signal.getType() instanceof CSignalType.STARTED)
    else if (signal.getType() instanceof CSignalType.RESUME)
    {
      // open resources
      m_fos = new FileOutputStream("/tmp/example.txt");
    }

    if (signal.getType() instanceof CSignalType.STOPPED)
    else if (signal.getType() instanceof CSignalType.SUSPEND)
    {
      // close resources
      m_fos.close();
    }
  }
}
```

Figure 4.80: Custom Signal Handling

Extensible Thread State Model. While legacy thread implementations, e.g. AWT threads, are not aware of the Java Thread Controller and can only be monitored, particular thread implementations offer the processing of control commands sent by the thread controller. Based on custom signal handlers, check points are regularly called by the thread and received commands are processed asynchronously, e.g. for preparing a thread to passivate. In this context, the presented approach allows to extend the legacy thread state model, as shown in Figure 4.81.

After a thread object has been created, the thread is situated in the thread state *new*. By calling the method `start`, the thread execution starts and the thread state changes to *running*. From there, a thread can change to the thread state *blocked*, e.g. by calling `suspend`, and back to *running*, by calling `resume`. Finally, a thread stops its execution when reaching the thread state *dead*, e.g. by leaving the method `run`. The presented approach extends this thread state model by introducing so called *check points* that are regularly called by the thread and check for extra state change requests, e.g. a *passivate* request for suspending the thread execution and saving the thread variables. In contrast to regular thread control methods, e.g. *suspend* and *stop*, which may intercept the thread execution anywhere in the executed code, check points are checked asynchronously by the thread itself, e.g. during a lengthy processed loop. The check point approach also allows developers to insert further handlers for preparing the thread to change the thread state, e.g. releasing acquired resources and monitors before terminating the execution.

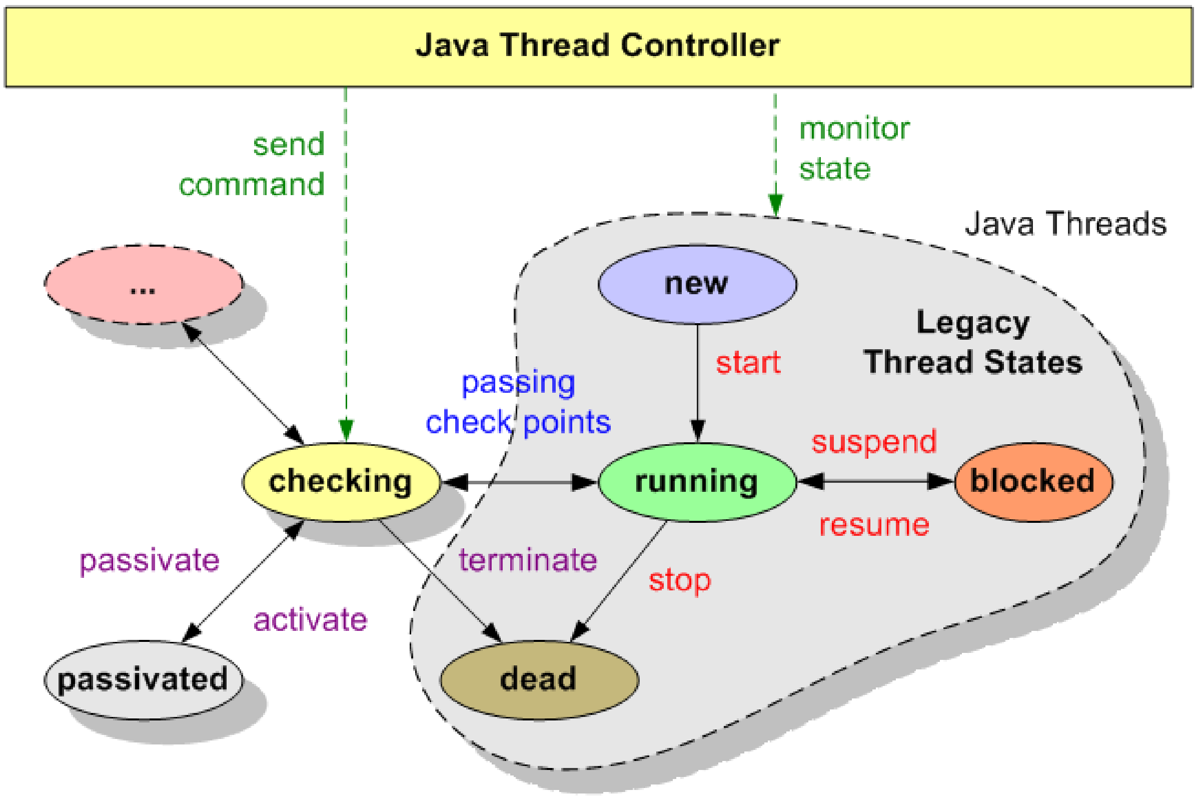


Figure 4.81: Extensible Thread State Model

4.7.1.5 Autonomic Application

The Java Thread Controller approach can be used to implement *Autonomic Task Execution* in a multi-tasking runtime scenario with not centrally managed threads of distinct applications, as shown in Figure 4.82.

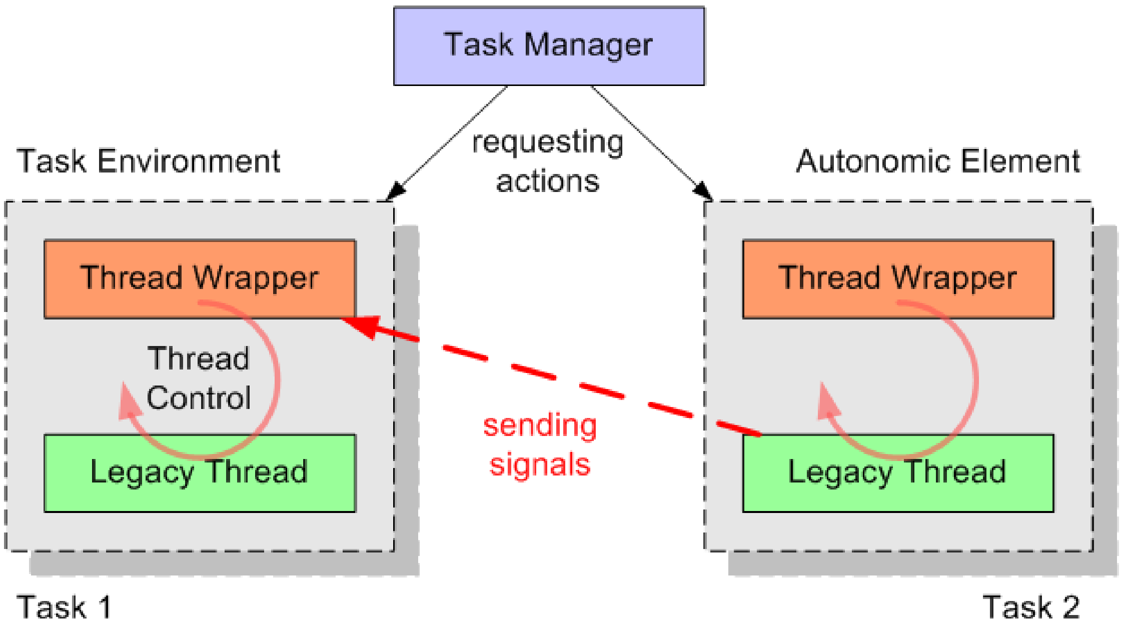


Figure 4.82: Autonomic Task Execution

A task environment groups all threads of a task and represents a program element to control the task execution by using a common task manager. The thread wrapper provides extra management functions above the legacy thread implementation, e.g. managing a signal queue for asynchronous thread interaction. If a task is created, the initial thread is linked to a new thread group and subsequently created task threads are automatically assigned to this thread group and dynamically associated with a thread wrapper.

The autonomic operation of the Java task environment is performed by monitoring and controlling the contained Java threads. The task manager tracks the creation of new threads and requests task actions by sending signals to the related thread wrappers. From this point of view, the task manager represents the requesting element, the thread wrapper acts as the autonomic manager, the legacy thread is the managed element and the task environment is the autonomic element. Similar to creating a regular component object [see Section 4.3.2], a thread component may be configured, contextualized and initialized during creation. For example, the task manager can pass an object reference to the task context and provide access to the current task environment *(self-configuration)*. During runtime, the task threads may be signalled by other tasks and have to process various signals. The thread wrapper can check the thread state and customize the signals, e.g. to improve responsiveness of the application by transforming one long execution suspension into many shorter ones *(self-optimization)*. A particular issue is the monitoring of the task to perform a certain action. For example, when a request is received to terminate a task all related threads have to be terminated before the task actually ends. The thread wrapper monitors the execution and may automatically kill certain threads, if they are not responding *(self-healing)*. Since the thread wrapper shields the actual thread object from external access, the approach can also be used to control access to sensitive threads in a multi-tasking environment, e.g. filtering and ignoring too many signals received from unauthorized or malicious tasks *(self-protection)*.

4.7.1.6 Related Work

The legacy JVM has been originally designed to support single application hosting and lacks basic multi-processing capabilities like Java task management [22]. In a multi-tasking scenario, every Java application is run in a separate JVM process. Since there is no built-in support for multi-process execution synchronization, a popular remedy is *thread-based multi-tasking* and to run multiple applications in separate Java threads. In this context, concurrent programming has not only to deal with application-specific thread execution but it has to consider cross-application thread control as well. In the following overview, various thread management solutions are regarded with particular respect to enable multi-tasking.

Legacy Java Virtual Machine. In a legacy JVM, an application may launch various threads to parallelize the program execution [258]. Over time, the runtime library was extended with sophisticated synchronization features to ease the implementation of concurrent workflows, e.g. by introducing the Java package `java.util.concurrent`. Particular application frameworks exploit these features to support multi-application execution and introduce custom programming models and control flows [203]. Popular examples are servlet containers like *Jakarta Tomcat* [16] and J2EE application servers like *JBoss* [179]. Since their application models have been designed to run applications at best without mutual interference, there is no support for thread management and execution synchronization across multiple applications. There are multi-tasking approaches that propose the extension of the legacy JVM with advanced multi-threading features. For example, *Fair Threads* [107] introduce a cooperative thread scheduling implementation which ensures the fair execution of multiple Java threads and allows event-based thread synchronization. A basic issue of thread libraries is the introduction of a proprietary thread programming model that does not support legacy thread implementations. Thus, existing application code has to be changed before it can benefit from additional thread scheduling and synchronization features.

Custom Java Virtual Machine. Multi-thread programming, in particular multi-thread synchronization and signalling, is a non-trivial development task [203]. In addition, the *cooperative thread scheduling* model of Java cannot be changed with Java libraries. Various approaches have been proposed to seamlessly extend the JVM with advanced thread management features while keeping the extra development effort moderate. For example, *MobileThread* [43] modifies the JVM and allows applications to inspect and control the internal thread states during runtime. From Sun, a *Multi-Tasking Virtual Machine (MVM)* has been proposed to explicitly support multi-tasking in a shared JRE without imposing developers to use a new thread programming model [75]. The MVM offers seamless execution of multiple legacy Java applications and can handle non-cooperative thread implementations. In this context, multi-tasking support is bundled with strong application isolation. Similar to third-party Java thread libraries, there is no common thread management and execution synchronization across distinct application instances. In general, custom JVM implementations may introduce advanced thread monitoring and scheduling schemes but they cause particular deployment efforts in a wide-area computing environment.

To summarize, the presented *Java Thread Controller* can be used with legacy JVMs and is deployed as part of the XDK. In contrast to related thread libraries addressing multi-tasking support, e.g. *FairThreads*, its focus is not the strong isolation of application threads but the *cross-application synchronization* in a shared JRE. The introduced thread management allows monitoring the execution state of concurrently running tasks and their threads. And though it may not access internal thread states like the custom JVM approaches, e.g. *MVM*, it offers advanced thread control and inter-thread communication features for cooperative Java applications that follow a particular thread programming model. As a result, the Java Thread Controller implementation may be used to monitor legacy threads in multi-tasking scenario and additionally it offers advanced thread synchronization features for custom thread implementations.

4.7.2 Java Execution Units

Due to the original design of the Java Virtual Machine to support the exclusive hosting of one application, the actual control of the application execution is missing in a multi-tasking setting. In this section, *Java Execution Unit (JEU)* is introduced to manage Java programs in terms of tracking the loaded code components, the instantiated data objects and the program execution. The realization of the approach is described and the application for ad-hoc execution migration is illustrated.

4.7.2.1 Motivation

There is an ongoing trend in Java programming to host various tasks in the same Java Virtual Machine and to benefit from the advantages of a multi-tasking environment, e.g. reduced overall memory footprint and shorter application startup time. For server-side applications, a well-known approach is to launch a Java application server like JBOSS and to run multiple tasks by loading program components into different host containers, e.g. Java servlets or EJB components. Similar approaches exist for running client-side applications where OSGI-based implementations have recently gained much attraction like the Eclipse Equinox runtime environment. Common to these approaches is the definition of particular deployment units like *Web Archives (WAR)* and *Enterprise Application Archive (EAR)* that usually contain a single application or service. In contrast, OSGI-based applications are typically fragmented into distinct bundle files that are dynamically resolved, loaded and composed one-by-one when the application is launched.

While an application framework greatly supports the dynamic composition and launching of applications, later on its task is typically limited on monitoring and controlling the application container. For example, a servlet engine like Jakarta Tomcat may shutdown a servlet container with the servlet therein and may relaunch an updated servlet release. However, there is no support to suspend a running servlet and proceed with the servlet execution from where it has been interrupted, e.g. resuming a lengthy task after a temporary server shutdown. A related issue is the need of tracking inbound and outbound object connections of the hosted applications to seamlessly reestablish resource bindings and remote object connections after resuming the execution, e.g. after being moved from one host container to another due to a migration request in a load-balancing cluster. In effect, there is a demand to handle suspension, migration and resumption of application execution in terms of relocating required code assemblies and instantiated data objects, restoring the current execution state and rebinding needed system resources; also known as *code mobility*.

4.7.2.2 Features

The overall goal of the *Java Execution Unit* approach is to manage the program elements of Java applications running in a multi-tasking JVM and to support code mobility in distributed computing scenarios. The major features are as follows.

Code Assembly Fetching. A Java Execution Unit (JEU) tracks the requested Java classes by using the Java Class Collection and Java Class Spaces approach. Whenever a task should be moved to another computing system or suspended to disk, the class collection configurations are serialized and sent along with the task data. To resume the task execution, the configurations are read and the required code assemblies are fetched by the target computing system from local code repositories.

Task Data Relocation. A JEU manages task data that is comprised of application data objects and thread elements, e.g. thread object, instruction pointer and stack variables. To move a JEU to another computing system, the threads can be suspended at the next check point to serialize the thread elements and the data objects. On the target computing system, the data objects are deserialized and the threads restarted from the last check points the execution was suspended.

Dynamic Object Reconnection. If a data object of the JEU has been bound to external objects by an inbound or outbound connection, the connections are dynamically restored when the JEU is moved. In case of remote object connections, e.g. via network links, the connection is reestablished without having the need to inform the remote partner in advance. In particular, inbound connections are automatically rerouted to the new object location.

Adaptive Resource Binding. Besides serializable data objects, a JEU binds system resources that cannot be transferred to another computing system or written to a file for later restoration, e.g. database connections or handles on GUI elements. This is resolved by implementing an adaptive resource binding approach that initializes suitable resources on execution resumption and restores resource bindings of the JEU before the task is actually restarted.

Regular Java Implementation. For the easy application of the approach in a cross-platform operating environment, the JEU implementation does not require a customized JRE but runs with any regular JVM, e.g. potentially installed on nowadays computing systems. Moreover, developers may add existing code components without modification or recompilation as long as all related threads can be suspended at well-known check points, e.g. when returning from a third-party library call.

4.7.2.3 Approach

The approach is based on *Java Execution Units (JEU)* that enable individual or all fragments of a running task to be moved from one computing system to another, as shown in Figure 4.83.

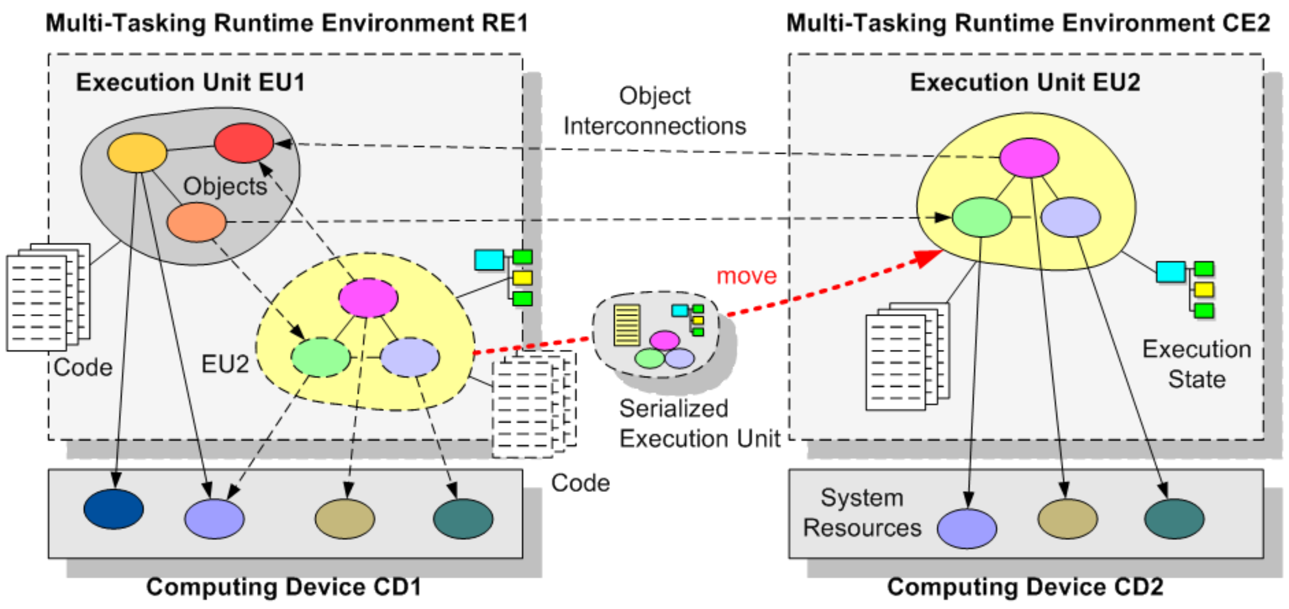


Figure 4.83: Serializable Execution Unit

For the definition of a JEU, various concerns of running a Java application in a multi-tasking runtime environment have to be regarded. First, an appropriately configured application environment has to be created in which the related program elements can be hosted [see Section 4.5.1], such as code assemblies and application objects. While running, the application may request various platform resources, such as a handle to the local graphical user interface [see Section 4.4.1], and establish connections to application objects on the same or a remote computing system [see Section 4.6]. Various execution threads can be forked which comprise altogether the overall task execution state [see Section 4.7.1]. From this point of view, a *Java Execution Unit (JEU)* is defined as a task fragment whose related code assemblies, platform resources, object connections and thread execution states can be independently tracked and managed from the rest of the task. A particular management feature is to suspend the execution at developer-defined check points in the program code and to serialize the related task elements in a way that the execution can be seamlessly resumed later. While being serialized, a JEU can be stored on the same computing system (hibernation) or may be transmitted to a remote computing system (migration). All information needed to create an appropriate application environment, rebind platform resources, reconnect remote objects and restore the execution state is contained in the serialized execution unit. Besides serializing a single task fragment only, entire tasks and user sessions may be moved in a nomadic computing scenario.

4.7.2.4 Realization

The realization of the JEU approach benefits from various XDK solutions concerning code deployment, resource bindings, object bindings and task management. The components of the approach are shown in Figure 4.84.

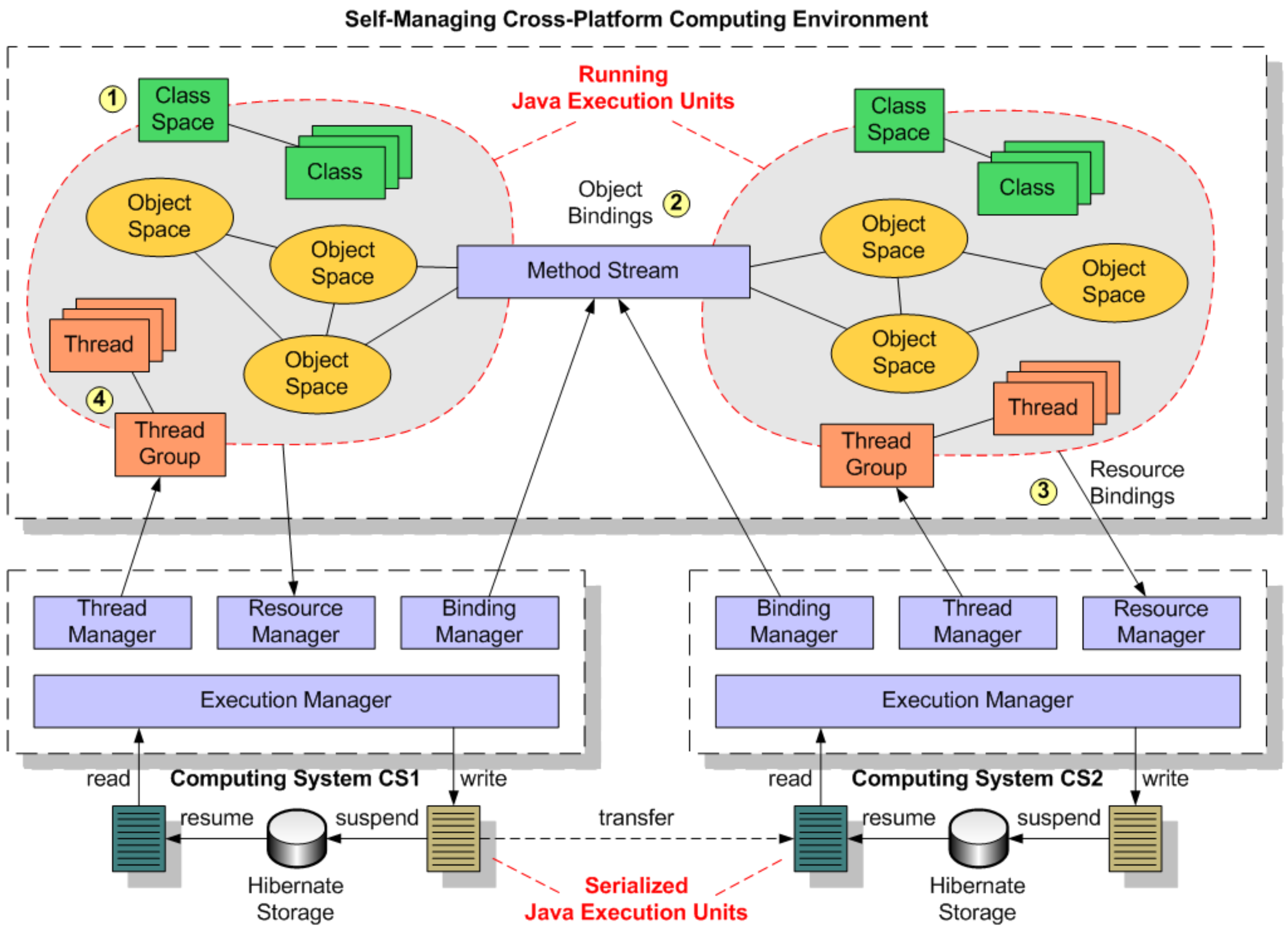


Figure 4.84: Java Execution Unit (JEU)

The code handling of a JEU (1) is based on *Java Class Collections* and *Java Class Spaces* as described in Section 4.2.2 and Section 4.3.1, respectively. In particular, the introduced ability to configure Java application composition without knowing the deployment scenario in advance allows serializing a JEU without including related code assemblies. In the context of execution migration, this is also known as *code fetching* which dynamically resolves required code packages on the target computing system in contrast to *code shipping* and transmitting the code packages. The object bindings to different object spaces on the same or a remote computing system are managed by using the *Java Remote Method Streaming* and *Java Object Spaces* approach (2) as described in Section 4.6.1 and Section 4.6.2. A particular feature is the individual tracking of virtual object binding and physical object communication that allows the seamless separation and insertion of JEU objects from and into an application environment, respectively. If a JEU is serialized and moved, all information about existing object bindings are added and used to reestablish the object communication on the target computing system. A related concern is the management of JEU bindings to platform resources (3), e.g. an application window in a graphical desktop system. Since a migrated JEU will usually not reopen a window on the originating host, the encapsulated application has to request a suitable application window on the currently hosting platform, e.g. by using the *Adaptive Resource Broker* approach as described in Section 4.4.1. Concerning the tracking of threads and their execution state (4), a JEU has to support the *Java Thread Controller approach* presented in Section 4.7.1. In detail, a signal handler has to be implemented that can process hibernation/migration signals to interrupt the execution of a JEU at given check points and serialize all application object and thread variables, as described below.

Serialization Workflow. A basic feature of the JEU approach is the transformation into a serializable form that can be easily transferred to another computing environment and transformed back. The serialization workflow of the underlying realization is shown in Figure 4.85.

If the execution manager receives a suspend request, the serialization is conducted as follows. By using the Java Thread Controller, a passivate signal is sent to all threads of the JEU that causes them to stop at defined check points and to save their execution state. After all threads have entered the passivate state, the execution unit starts to lock all object skeletons and outbound connections to its objects as well as the object stubs and inbound links by using the binding manager. The connections are closed and the execution unit returns control to the execution manager. At this point, the execution unit is no longer executed and all object connections have been locked. Now, all application objects of the JEU, including thread objects and object bindings, are serialized into a byte stream and can be easily managed, e.g. stored in a hibernate storage for later resumption or transmitted to a remote computing system. To resume the execution, the serialized JEU is passed to the execution manager that reads in and initializes all application objects. In particular, the thread objects use the saved attributes to restore their execution state for restart. By using the binding manager of the Java Remote Method Streaming approach, the object stubs and skeletons are transparently reconnected with their original endpoints and unlocked. Finally, the Java Thread Controller restarts all threads by sending appropriate activate signals and the execution unit returns back to work. The details of injecting a JEU into an application environment and initializing the object data is as follows.

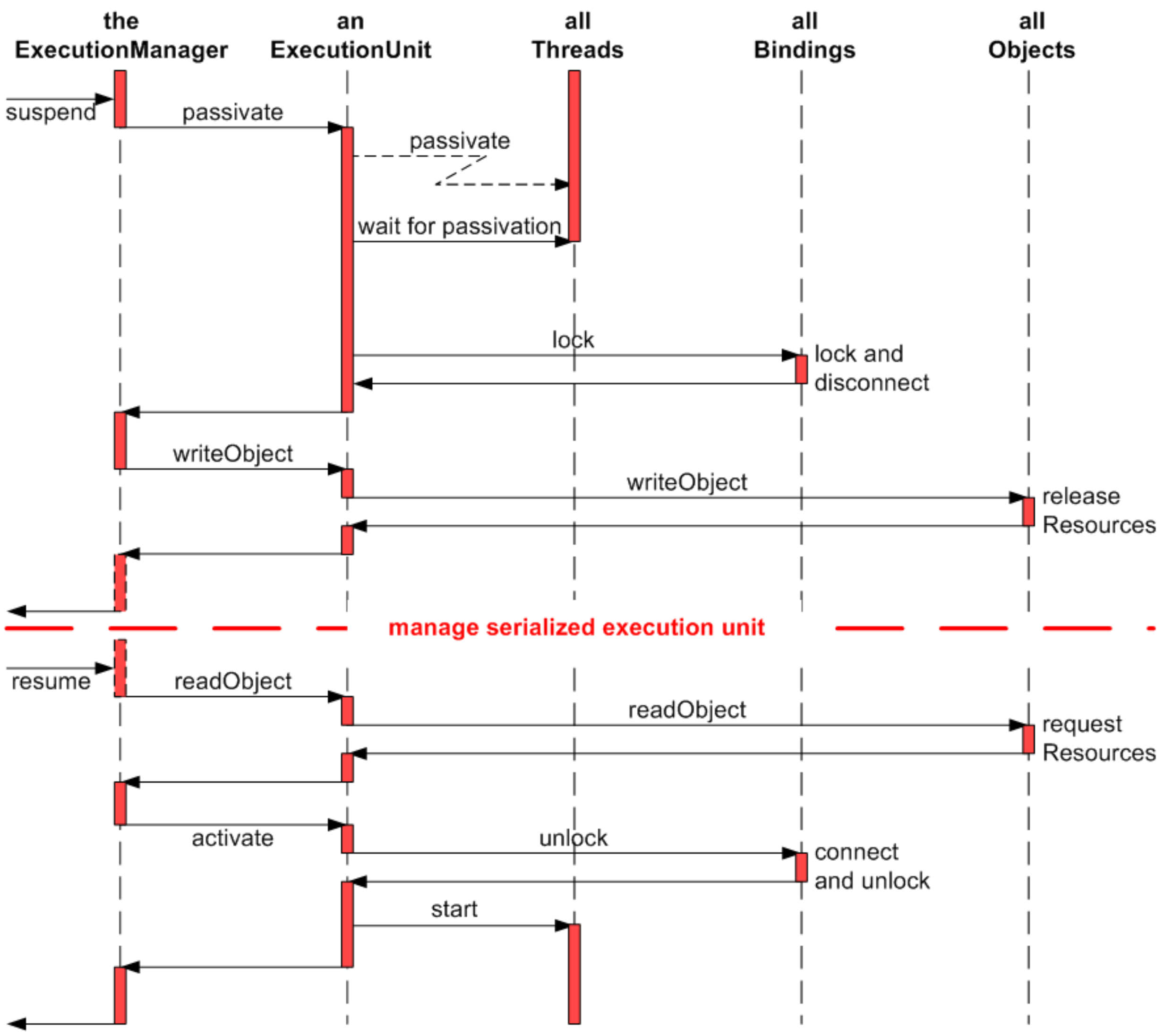


Figure 4.85: Serialization Workflow of an Execution Unit

Unit Contextualization. A particular issue for launching an application is the dynamic provision of required hooks for requesting bindings to environment and platform resources. The XDK implementation uses an *Inversion-Of-Control (IOC)* approach and checks the application for implementing various features interfaces [see Section 4.4.1] to trigger the contextualization, as shown in Figure 4.86.

The application `CExample` implements the feature interfaces `IContextualizable`, `IInitializable` and `IPassivatable` to allow the application environment to call related control methods. Concerning the resumption of a serialized JEU, the example shows a method `contextualize` that is called several times to pass various context hooks like a scene context, a thread context and a task context. In general, a context hook represents a specific program interface to a related resource object of the hosting application environment, e.g. for interacting with the application task object or requesting platform resources, as described below.

```
public class CExample
                  implements IContextualizable, IIniatilizable, IPassivatable
{
  public void contextualize(IContext ctxt)
  {
    if (ctxt instanceof ISceneContext)
    {
      m_sceneContext = (ISceneContext) ctxt;
    }
    else if (ctxt instanceof IThreadContext)
    {
      m_threadContext = (IThreadContext) ctxt;
    }
    else if (ctxt instanceof ITaskContext)
    {
      m_taskContext = (ITaskContext) ctxt;
    }
  }
  ...
```

Figure 4.86: Contextualization of Execution Unit

Execution Initialization. After the JEU has been contextualized, the execution manager checks for the next feature interface `IInitializable` and the existence of the related method `init`, as implemented in the example in Figure 4.87.

```
  public void init()
  {
    IResourceManager resourceManager = m_sceneContext.getResourceManager();
    m_desktopManager = (IDesktopManager)
                 resourceManager.requestResource(DESKTOPMANAGER_RESOURCE_ID);

    m_desktop = m_desktopManager.requestDesktop();
    m_windowPanel = (IWindowPanel) m_desktop.requestResource(WINDOWPANEL_ID);

    m_appWindow = new CClientFrame(m_windowPanel);
  }
  ...
```

Figure 4.87: Initialization of Execution Unit

The method `init` can be used to perform specific initialization steps before the actual use of the object and its application. The pendant of `init` is the method `exit` that is called before the application is terminated. In the context of initializing a JEU, an object can implement `init` to request the graphical desktop manager for creating a client frame window.

Custom Object Passivation and Activation. In addition to the object initialization of a regular application start, JEU objects may have to save and restore specific runtime data for the seamless restart of the execution. The feature interface `IPassivatable` decorates objects that provide related methods to be called before passivation and activation, as shown in Figure 4.88.

```
public void passivateObject()
{
  m_lastFrameBounds = m_appWindow.getInternalFrame().getBounds();
  m_lastTextFieldContents = m_appWindow.getCurrentTextFieldContents();
}

public void activateObject()
{
  m_appWindow.getInternalFrame().setBounds(m_lastFrameBounds);
  m_appWindow.setCurrentTextFieldContents(m_lastTextFieldContents);
}
```

Figure 4.88: Custom Object Passivation and Activation

In the example, the method `passivateObject` is used to save the attributes and content of the application window. Later, they are serialized as any other object data and restored during deserialization. To activate the object before restart, the method `activateObject` is called, and the attributes and content of the application window are restored.

Object Action Stream. While well-known application objects like the one containing the starting point can be easily accessed by the application environment, other application objects are not directly reachable, e.g. referred via various links of distinct object instances. In fact, the execution manager can not directly call custom object passivation and activation methods of all application objects as required by the presented approach. This problem is solved in the realization by customizing the original Java serialization implementation and introducing so called object action streams, as shown in Figure 4.89.

```
public class CPassivationStream extends ObjectOutputStream
{
  ...

  protected Object replaceObject(Object obj) throws IOException
  {
    if (obj instanceof IPassivatable)
    {
      ((IPassivatable) obj).passivateObject();
    }
  }

  ...
}
```

Figure 4.89: Object Action Stream

Normally, the original Java serialization implementation is used to persist all serializable objects that can be reached by traversing regular object references (persistence-by-reachability). The Java compiler generates the methods `readObject` and `writeObject` method to read and write the object attributes, and to traverse through all reachable objects. This particular traversal implementation is exploited by the object activation stream to call specific methods on the objects coming across the serialization. In Figure 4.89, the `CPassivationStream` is an `ObjectOutputStream` that is used to "persist" the JEU objects. However, instead of writing the object attributes the method `replaceObject` is overridden to call the method `passivateObject` on implementing objects. As a result, an object output stream pretends to serialize all linked application objects but in fact, it performs a specific action by calling related methods on the application objects.

Performance Evaluation. Performance measurements exhibit that the inherent migration overhead for passivating and activating regular application objects is small compared to the time for transmitting the serialized execution units across the network, as shown in Figure 4.90.

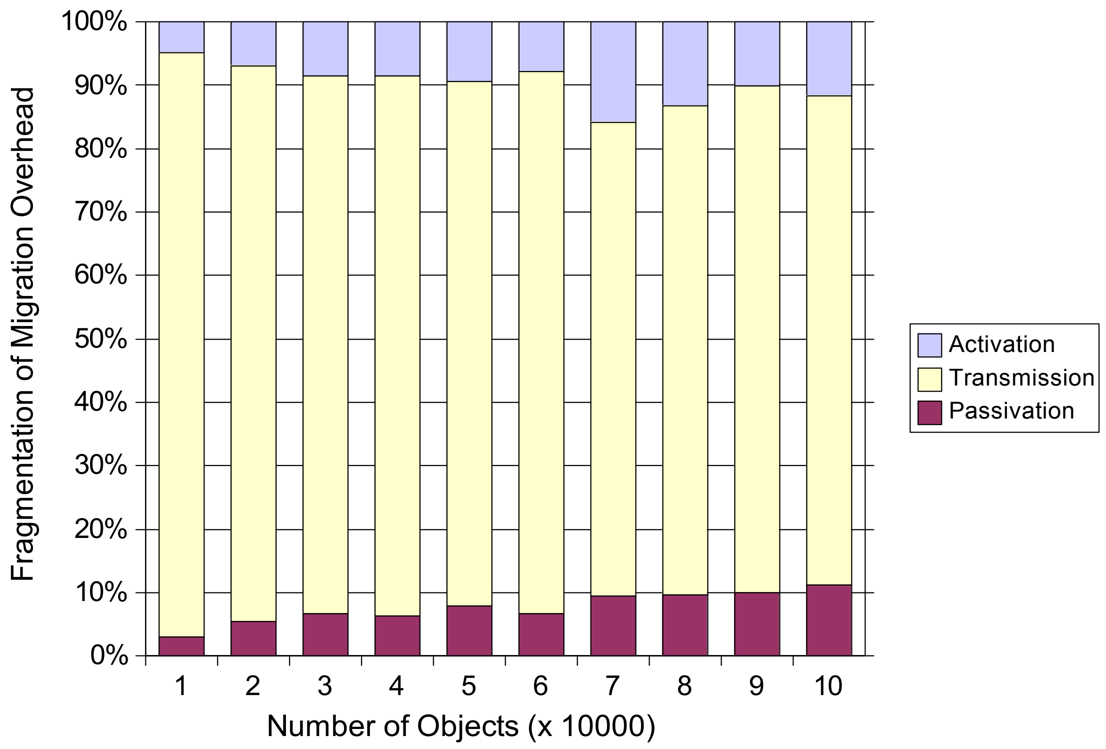


Figure 4.90: Fragmentation of Migration Overhead

In an exemplary test bed to determine the fragmentation of the migration overhead, a running application is moved from one computing system to another one. Both systems are located in the same network, run a legacy JRE and use a common crosslet repository. At the beginning of the migration process, the application and its threads are passivated, all related application objects are serialized and transmitted to the target node. Then, required code assemblies are fetched from the crosslet repository and the application objects are deserialized. After that, the migrated thread objects are restored and the thread execution is resumed. The relative overhead appears to be almost constant over a wide range of object counts whereby the activation needs more time than the passivation. This is due to the recontextualization and the reinitialization of the migrated objects after the deserialization into a new runtime environment. In practice, the times heavily depend on the application scenario, e.g. initially fetching related code components, dealing with system resources or reconnecting remote object bindings may delay the migration for an indefinite time.

4.7.2.5 Autonomic Application

The Java Execution Unit approach can be used to implement *Autonomic Load Balancing* on interconnected application execution engines running in a cross-platform peer federation, as shown in Figure 4.91.

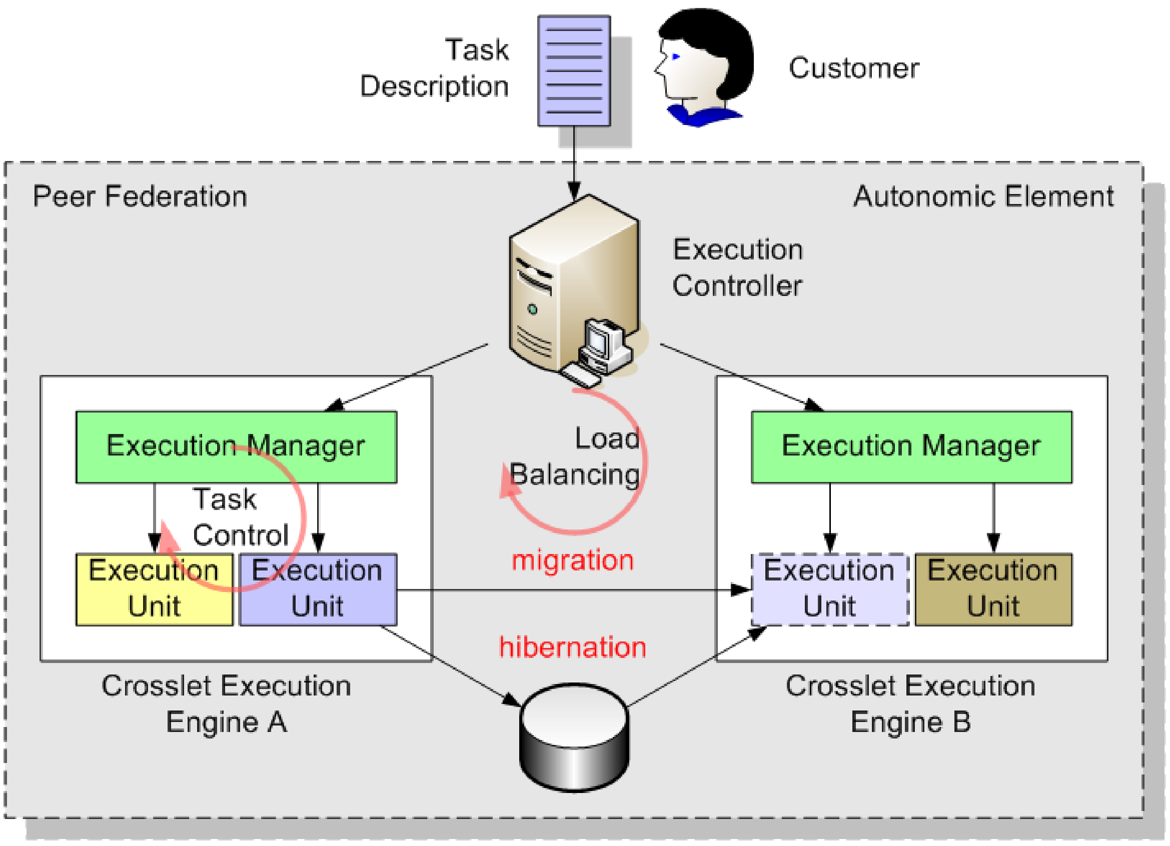


Figure 4.91: Autonomic Load Balancing

Based on the application execution engine approach introduced in Section 4.5.1, a customer can issue tasks to a peer federation by sending a task description to the execution controller that monitors and controls various peers. The execution controller determines the current load distribution and selects an appropriate application execution engine to process the task, e.g. a peer for which crosslets to run the related execution units can be found and installed. In this context, an application execution engine can host several execution units and in case a peer gets overloaded, the execution controller can migrate a single execution unit to another peer, or temporarily suspend the related task processing and hibernate the task to disk for a while. The autonomic operation is performed by controlling the task dispatching and monitoring the execution state of every peer. From this point of view, the customer acts as the requesting element, the peer federation represents the autonomic element, the execution controller is the autonomic manager and the controlled application execution engines are the managed elements. The peer federation is comprised of separately managed peers that may join and leave the federation at any time. A particular problem is the selection of an appropriate peer on which the crosslets needed to process the task can be deployed. The execution controller queries the execution managers and determines the best match for deploying a task according to the platform capabilities, task requirements and deployment policy, e.g. using a high-performant machine for business customers and filling up the remaining time with regular customer tasks *(self-configuration)*. Since the composition of the peer federation and the arrival of new tasks are ever-changing, a particular issue is the dynamic balancing and migration of tasks across available peers, e.g. according to a given plan to equalize the CPU load or minimize the costs for processing a task *(self-optimization)*. Next, it may happen that a peer is no longer available and the task processing is unpredictably aborted. The execution controller will notice this and try to redeploy the task to another peer. In this context, the execution controller can instruct a peer to periodically hibernate the execution unit in a central task repository for being able to relaunch the task on another peer *(self-healing)*. Finally, the execution controller monitors the overall condition of the peer federation to protect distinct peers to get overloaded. Similarly, new tasks may be queued before deployed for processing *(self-protection)*.

4.7.2.6 Related Work

Over time, various notions of *code mobility* have been introduced in the literature. A good starting point on the terminology and issues of code mobility can be found in [126]. Another paper focuses on process migration and details the complexity of transferring a process from one computing system to another [241]. The support of mobile objects in Java is illustrated in [246]. As a result, a possible definition of code mobility is the capability to dynamically change the bindings between code fragments and the location where they are executed [52]. In the following section, the concerns of code mobility are discussed with respect to execution migration, weak and strong mobility as described in [126]. A common view is the separation of concerns into data objects, code fragments and execution state, as illustrated in Figure 4.92.

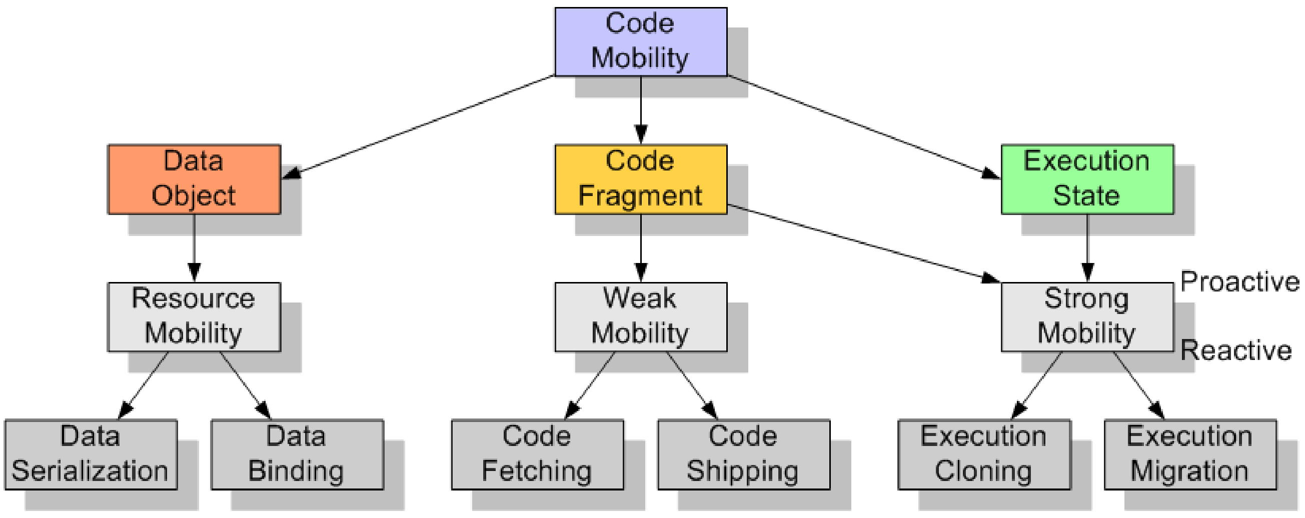


Figure 4.92: Concerns of Code Mobility

A basic concern is the transfer of *data objects* and the handling of references from and to mobile objects. Object links are separated due to migration and must be reestablished later. Another issue is the suitable selection and binding of local system resources when an execution unit moves, such as the window manager of a GUI framework or a particular service of the operating system. Another concern is the provision of needed *code fragments* when executions units are moved. The selection and retrieval of suitable code packages as well as their composition may differ among various hosts. A particular issue is the configuration of the target computing environment to run the execution unit, such as the path to locally installed programs. The migration of a running execution unit has to intercept the processing in the current computing environment and continue with its operation after the migration. To this end, the *execution state* is recorded on the source node, relocated and restored on the target node. A particular problem is the reading and transfer of transient data, such as local variables within a method call. From these concerns, various categories of code mobility can be deduced. They arise from different mixtures of features and mainly differ in what is actually moved from one computing environment to another. The first category is *resource mobility* that refers to the seamless serialization of data objects along with the code fragments needed to reinstantiate the object. On the other hand, bindings to resources that cannot be moved are encapsulated by a suitable connection approach. Related stub classes are deployed on the target node and object references are created that actually point to a stub that in turn connects back to a skeleton on the original host. Similar to that, remaining objects reconnect to migrated data objects the other way round. A second category is *weak mobility* where actually only code is transferred; possibly with some initialization data. The execution initially starts on the target node and there is no connection to other execution units or computing environments. No data is serialized and there are no object references that have to be maintained. Basically, the code is either shipped with the initialization data or autonomously fetched by the target node, e.g. from a remote code repository or a local code archive. The third category is *strong mobility* which addresses the seamless movement of an execution unit and represents the most complex form of mobility. Execution migration requires everything needed to run an execution unit to be transferred from one computing environment to another. This includes data objects, object references, code fragments and the current execution state of each thread and process. A particular variant is execution cloning that lets the original execution unit continue and consequently results in two separately executed units. If the migration of a code component is externally triggered, this is called *reactive migration*. If the code component itself decides when to migrate, this is called *proactive migration*. As a result, the level of complexity increases from resource mobility via weak mobility to strong mobility. With respect to ad hoc execution migration, the support of strong mobility requires an approach that is able to asynchronously signal a migration request and execution units that are able to handle this request at any time. There are several proposals that deal with strong mobility and ad hoc Java execution migration. The approaches mainly differ in the way how they address the regarded concerns and they can be separated into the following categories:

Modified Java Virtual Machines. Modified and custom Java Virtual Machines (JVM) allow including advanced features that are not available in a standard JVM. As an example, the permanent monitoring of the execution state and the ability to inspect the calling stack greatly support *strong mobility* down to the interception and migration of a running thread nearly at each line of code. Various approaches benefit from a custom JVM, such as *MVM* [75], *MobileThread* [43], *JavaThread* [47], *Camel* [6], *DGET* [101], *NOMADS* [349, 256] and *JESSICA2* [396, 397]. Detailed reviews can be found in [54]. According to these reviews, a major disadvantage is the lack of portability in many ways which affects the application of the approaches in large-scale heterogeneous environments. Above all, the use of a custom JVM requires a separate installation process that presumes that administrators are willing to trust a non-certified Java sandbox approach and also to keep it up to date with new Java releases and features, e.g. Java language extensions and JIT compilers. Another drawback is their confinement to handle the execution state. They do not consider separated object links, different platform configurations and application requirements in a heterogeneous environment.

Java Code Instrumentation. Another category uses custom instrumentation tools that rely on a standard JVM but generate modified byte code to enable *strong mobility*. Extra code is inserted that allows capturing the execution state of the mobile components and tracking object references that must be restored after migration. Some approaches, such as *WASP* [127] and *X-KLAIM* [33], are based on source code modifications during design time and operate like a Java preprocessor to translate strong mobility into weak mobility. A different option is to transform existing Java byte code during runtime, such as in *Correlate* [358], *JavaGoX* [311], *JMobile* [299] and *JADE* [228]. Next, the Java platform debugger may be used to inspect internal data structures, such as proposed in the *Collaboration and Coordination Infrastructure for Personal Agents (CIA)* [173]. Common to all these approaches is the generation of standard compliant Java applications that may be run by a legacy JVM. However, they presume the application of the same tool on all participating nodes and may be not mixed with another instrumentation approach, e.g. aspect-oriented programming. They also lack specific support for code deployment and binding of migrated data objects in a cross-platform operating scenario.

Mobile Code Frameworks. The next category is related to mobile code frameworks that can be used with every standard compliant Java development and runtime environment. The frameworks typically introduce object middleware layers and specific programming models to support object relocation and code mobility. Since they are not designed to inspect the calling stack or monitor the actual thread execution, they at best enable *weak mobility*. For example, both *MobileRMI* [20] and *Mobile Object Workbench* [50] are based on a distributed objects approach. They use a middleware layer that allows the automatic updating of remote references to mobile objects and the location-transparent movement from host to host. *Mikado* [34] provides a mobile code framework to implement mobile agent systems by supporting the transparent shipping of code components during the migration. In *Kalong* [44], an advanced code shipping approach is introduced to reduce migration overhead by dynamically determining which Java classes are actually required on the target node [194]. In this context, particular security issues arise from executing dynamically received program code, e.g. sent by a malicious host [56, 392]. Various approaches emerged to address mobile code security, e.g. by using a sandbox, signing shipped code components or checking code with *Proof-Carrying Code (PCC)* techniques like the Java Byte Code Verifier [308]. Concerning the adaptive migration of mobile objects, *Ajents* introduces a distributed Java software infrastructure for parallel computing that is able to create objects on remote hosts. [177]. A related mobile code system is *POEMA* that enables the policy-based dynamic reconfiguration of mobile code applications [244] and separates the mobility concerns from application functionality [243]. It supports different mobility pattern, such as remote evaluation, code-on-demand and mobile agent, and allows reconfiguring the mobile code application by using Java reflection and the policy language *Ponder* [79] In *FarGo*, the application code is divided into mobile modules, called *complets*, and a scripting language is used to specify the dynamic layout strategy at loading time [168]. Although these mobile code frameworks may be used in a heterogeneous environment, they are not feasible for large-scale and separately managed installations. They either rely on unsecure code shipping or use a static deployment approach that does not support different class repositories, such as [34]. Furthermore, focusing on a single communication approach, such as legacy *RMI* [20] or *CORBA* [319], prevents the ad hoc use of different connection solutions suitable for the capabilities of the currently involved computing environments.

In summary, there are many approaches addressing code mobility and supporting *ad-hoc execution migration* to some extent. Depending on the capturing level, resource mobility, weak mobility and strong mobility can be distinguished which roughly correspond to the migration of application data, application code and application state. Actually, full support of transparent execution migration is offered by strong mobility approaches only. In this context, the most difficult task is to capture and restore the application state, e.g. by tracing the thread execution and method calling stack. This can be only achieved by either modifying the hosting JVM, injecting extra monitoring code into the application or instrumenting the Java platform debugger. While modified JVM approaches, e.g. *JESSICA2*, can be applied to legacy Java code, they cannot be used easily in a large-scale cross-platform operating environment due to the custom JVM installation requirements. Approaches that modify existing application code during compile time, e.g. *X-KLAIM*, cannot be applied in dynamic application composition scenarios. Against this background, mobile code frameworks like *Mikado* promise to enable transparent execution migration for applications following a particular programming model. From this point of view, the *Java Execution Unit* approach can be seen as a mobile code framework that supports ad-hoc execution migration by introducing a proprietary application programming model. In contrast to related work, it does not rely on static code deployment in advance or the shipment of migrating code. Instead, it utilizes the crosslet deployment model to offer adaptive code fetching. Thus, the execution units reduce the amount of migrating data and can select appropriate code components on the target computing system, e.g. by rebinding a compatible third-party library in favor to download the original library release. In addition, characteristic security issues of mobile code systems can be particularly addressed, e.g. by configuring trusted crosslet repositories from where missing code components can be exclusively fetched.

4.8 Summary

In this chapter, the *Crossware Development Kit (XDK)* has been presented which follows the idea of building an *Autonomic Cross-Platform Operating Environment (ACOE)* as presented in Chapter 3. It is completely written in pure Java and does not use any custom development tools or a modified JVM. It provides various software components to address the objectives of the ACOE listed in Section 3.1.3 and delivers a Java library that can be linked and deployed with an application installation. In the following discussion, the distinct features of the XDK implementation are briefly summarized and their practical feasibility is highlighted.

Distributed Code Deployment. Concerning distributed code deployment, the XDK supports assembly packaging and deployment by introducing *Self-Descriptive Crosslets*. The use is not limited to Java applications and can be used to deploy and resolve assemblies in a self-managing way. In particular the setup of remote crosslet repositories facilitates the distributed code deployment. Though the creation of crosslet archives (XAR) is quite simple, it does not follow a common packaging standard, e.g. to describe the assembly content and dependencies. Moreover, crosslets can be effectively used with an XDK based implementation only. Related to Java code deployment, the XDK also introduces *Java Class Collections* that virtually groups Java classes and allows adaptive class resolution by evaluating class collection queries. By using a standard class loader approach, regular Java applications can be transparently composed and launched without code modification. This is not valid, however, for application containers already managed by a custom class loader like in a Java servlet environment.

Dynamic Software Composition. Another feature of the XDK is the support of dynamic software composition by introducing the *Java Class Spaces* approach. It allows organizing the class loaders in a shared application environment and can be transparently used with legacy Java applications. In combination with class collections, class spaces may be organized to share common classes and to enable the selective reloading of application classes. For the same reason, class spaces may be only used as long as no custom class loader is installed and therefore the approach is not usable to organize existing application containers. To separate the virtual application composition from the physical component deployment, the XDK introduces *Java Loadable Modules* that represents initializable and queryable application composition units. On the one hand, the approach hides the actual component instantiation from the application by managing the module dependencies and life cycle. On the other hand, an application developer has to use the XDK module manager and extend the application code; hence the approach cannot be used out-of-the-box.

Shared Application Hosting. A further advancement concerning resource resolution in a multi-application hosting environment has been introduced with the *Java Resource Binding* approach. An adaptive resource broker mediates the mapping of application roles on component resources and decouples the tasks of application and component developers. The crucial elements are the description files that are evaluated by the resource broker during runtime. In practice, this approach requires much effort to create the role and resource descriptions. Moreover, a particular application model is used that prevents the use of legacy libraries. A related development is the *Java Task Space* approach to organize task related resources in a multi-session environment. Stages are introduced to manage system and application resources while scenes deal with task objects. The handling of task spaces is quite complex and the effort to manage the resource interactions is only worth for collaborating sessions in a shared application environment. For the common shielding of application instances it is not needed.

Pervasive Environment Customization. A major XDK approach is the introduction of an *Application Execution Engine* that represents a self-contained application launcher and task processing unit. Based on a task description, a matching application is selected by a task manager and the related application requirements are passed to a runtime manager that prepares a suitable application environment. Though the approach also works for native applications, it relies on the proper description and adjustment of task, application and runtime configuration. In practice, this effort makes it difficult to extend the application execution engine to new tasks and applications. Concerning pervasive environment customization, the XDK introduces the *Roaming User Profile* approach to synchronize user preferences across distributed computing systems, e.g. in a nomadic use case. After user login, a local profile manager connects to a remote profile repository and deduces the effective user preferences for a new computing system, e.g. by evaluating existing user preferences. Regular Java applications can easily interact with the profiles by using the standard Java preferences API. The disadvantage, however, is the simple synchronization approach that exchanges the entire profile and slows down the login procedure.

Virtual Object Interconnection. The XDK supports transparent object interconnection across network boundaries by introducing *Java Method Streams* and separating the real network communication from the virtual object communication. In contrast to legacy network middleware approaches, the method stream implementation offers developers a protocol-independent API to bind remote objects and call methods in a heterogeneous and changing networking scenario. This is facilitated by using Java dynamic proxies and Java reflection that shield the application from network connection issues. Though the method stream approach eases the object communication for pure

Java environments, it is not usable in a cross-language scenario. For multi-tasking environments, the *Java Object Space* approach extends the method streams implementation to isolate Java object groups hosted within the same JVM by separating the object references and the related object instances. It enables the control of inbound and outbound communication of encapsulated objects that can be used to serialize distinct applications. Since the approach relies on the particular use of custom class loaders and Java reflection, it cannot be used with legacy application containers, e.g. in a Java servlet environment.

Ad Hoc Execution Migration. While the Java object space approach addresses the application data, the *Java Thread Control* implementation of the XDK provides features for asynchronous thread signalling and synchronous action handling. This is achieved by particular thread wrappers and check points in the application code. In fact, every time an application passes a check point, it checks for waiting thread signals using a callback in the thread wrapper. Since the approach introduces a custom thread wrapper class that has to be explicitly used by the application developer, it cannot be used for legacy Java code. Moreover, the implementation assumes the cooperation of the addressed applications and requires particular design and implementation effort. A particular feature of the XDK is the introduction of *Java Execution Units* to encapsulate the program elements of running Java applications and enable their ad-hoc migration in a distributed computing environment, e.g. due to load-balancing requests. The core idea is to control the resource binding, object connections and task execution of an application by using the object space, method stream and task control features presented above. An application developer extends the migrating object with corresponding callbacks that are called during passivation and activation of an execution unit. While the migration implementation has proven its feasibility, it is only usable with specifically extended applications. In this context, the XDK only supports weak code mobility and cannot imperatively interrupt a running thread until it passes the next check point.

Overall, the XDK implementation represents a self-managing integration middleware that addresses deployment, composition, hosting, customization, interconnection and migration of Internet applications in an autonomic cross-platform operating environment. In this context, the focus of the XDK is the provision of a self-managing software infrastructure. The concrete implementation of advanced autonomic policies, e.g. by introducing utility or goal-based policies, has been not yet addressed. The same is valid for common security concerns in a distributed environment, e.g. denial-of-service attacks and malicious code deployment. Finally, a major disadvantage is the proprietary Java realization that makes it difficult to use and integrate the XDK in legacy application systems, e.g. an OSGI or servlet container.

5. ODIX - The On-Demand Internet Computing System

5.1 Introduction

In this chapter, the realization of the *On-Demand Internet Computing System (ODIX)* for supporting *On-Demand Internet Computing (ODIC)* described in Chapter 2 is presented. First, the goals, approach and system architecture are outlined. Then, the ODIX components, namely On-Demand Application Engine, Internet Application Workbench, Internet Application Factory and Internet Application Federation, are described. Various parts of this work have been already published in [274, 275, 279, 281, 283, 284] and carried on in [66, 235, 318, 382]. The section ends with a summary about the suitability of ODIX to perform On-Demand Internet Computing as presented in Section 2.4.

5.1.1 Goals

The overall goal of ODIX is to enable on-demand task processing following the *shift from resource-centric to task-centric computing*, as outlined in Section 2.4.1. In particular, required applications for processing a task should be retrieved, installed and run without particular user intervention, as shown in Figure 5.1.

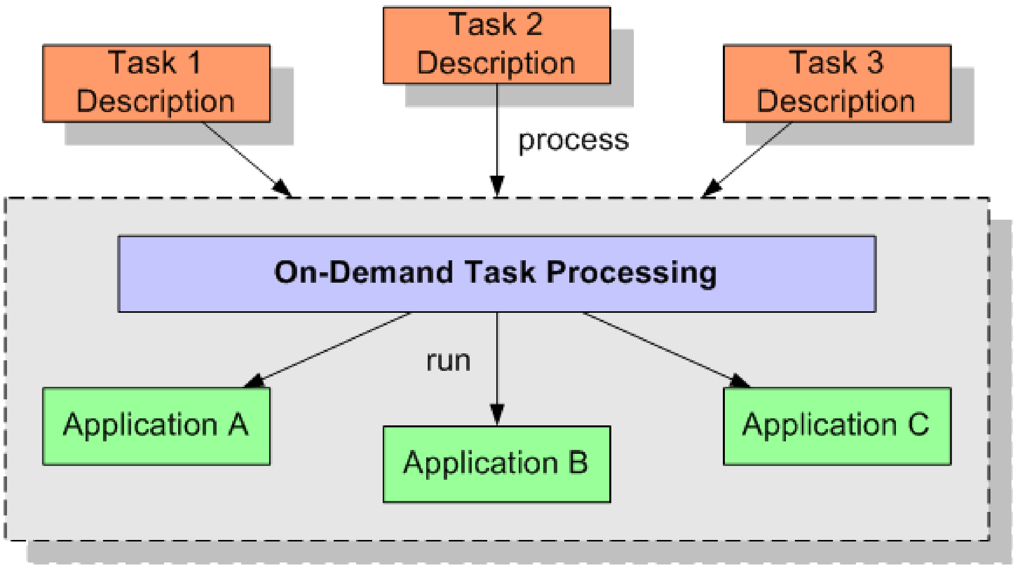


Figure 5.1: On-Demand Computing System

A task description does not contain any references to the application needed to process the task but entails task-related configuration statements only, such as the task type and where to find the task data. The actual selection of the application is dynamically performed by ODIX with respect to the task description and the current runtime scenario, e.g. capabilities of the employed computing platform, already deployed applications and present user preferences. Based on the vision of On-Demand Internet Computing (ODIC) illustrated in Section 2.4.3, the support of various task processing schemes are pursued by ODIX, as follows.

Local Task Processing On-Demand. In local computing scenarios, a user employs a local computing device to process a task, e.g. editing a document or retrieving his or her emails on a desktop computing system. ODIX should support the local task processing on-demand by providing a *pervasive computing environment* to run related applications and which customizes itself to his or her needs. A specific challenge is the alternating use of potentially heterogeneous and separately managed computing devices, e.g. at home and at the office.

Remote Task Processing On-Demand. In remote computing scenarios, a user or an application engages a remote computing device to deploy and run a task, e.g. uploading a media asset for analyzing and indexing the meta data. ODIX should support the remote task processing on-demand by providing a *shared computing environment* to utilize provided computing resources with concurrently running applications. A particular challenge is the handling of multi-session task processing, e.g. shielding private data while sharing common resources.

Distributed Task Processing On-Demand. In distributed computing scenarios, various computing devices are bundled to process complex and lengthy tasks, e.g. for analyzing large collections of measuring data. ODIX should support the distributed task processing on-demand by providing a *federated computing environment* to enable the seamless interaction and collaboration of networked computing devices. Particular challenges are the changing composition of the federation and the lack of a central authority to rule the computing nodes.

5.1.2 Approach

The ODIX approach is based on the self-managing operation of a computing system to prepare required application environments for launching Internet applications on-demand. In this context, the realization greatly benefits from the features of an *Autonomic Cross-Platform Operating Environment (ACOE)* as implemented by the *Crossware Development Kit (XDK)*, presented in Chapter 3 and Chapter 4, respectively. The resulting system architecture of ODIX is shown in Figure 5.2.

On top of the XDK, the ODIX implementation addresses the on-demand task processing goals presented in Section 5.1.1 and is comprised of several components, as outlined below.

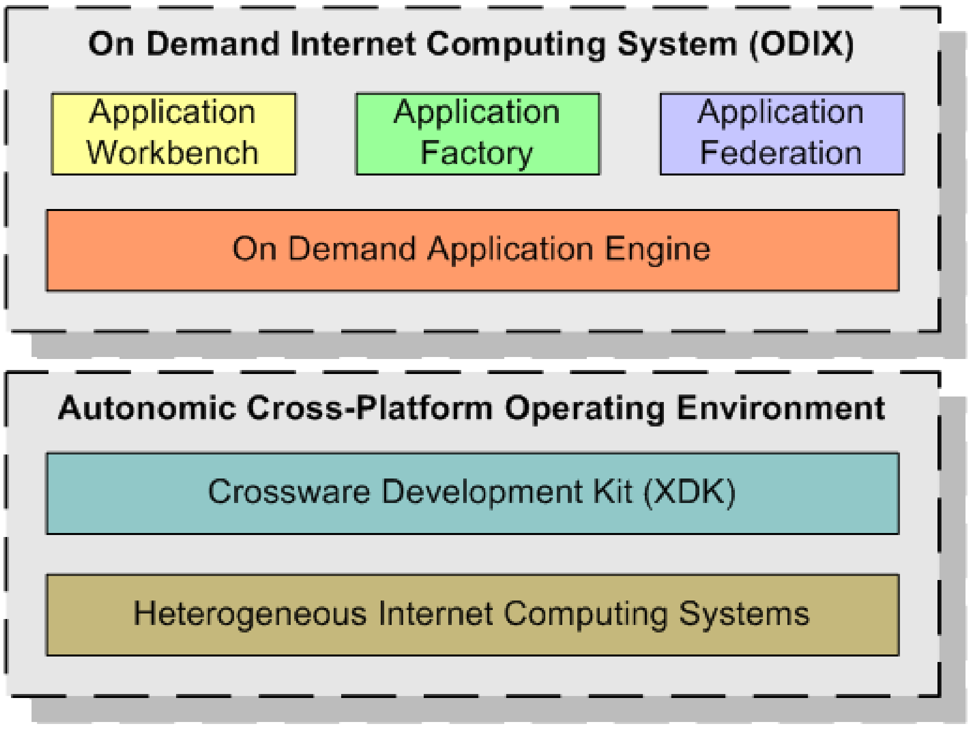

On Demand Internet Computing System (ODIX)
Application Workbench
Application Factory
Application Federation
On Demand Application Engine
Autonomic Cross-Platform Operating Environment
Crossware Development Kit (XDK)
Heterogeneous Internet Computing Systems

Figure 5.2: On-Demand Internet Computing System (ODIX)

On-Demand Application Engine. A core component of ODIX is the *On-Demand Application Engine* that has to be initially installed on the task-processing computing system. It implements the basic functions for launching distinct Internet applications on-demand, for managing concurrently processed tasks and for integrating the overall task processing into the currently engaged computing system, e.g. by selecting and using locally installed native executables for processing a task.

Internet Application Workbench. For local task processing on-demand, the *Internet Application Workbench* can be dynamically retrieved and installed on the currently employed desktop computing system. While it mimics the operation of a native GUI desktop like MS Windows, it likewise supports the seamless integration of the task processing into the installed desktop operating system, e.g. by using legacy application windows of the graphical desktop interface.

Internet Application Factory. For remote task processing on-demand, the *Internet Application Factory* is typically installed on a computing system in advance and is meant as an application server for running dynamically deployed tasks. It provides a network service interface to control the task processing using a remote computing device while supporting multi-tasking operation at the same time, e.g. sharing the computing device with several sessions.

Internet Application Federation. For distributed task processing on-demand, the *Internet Application Federation* enables the self-organization of networked peers and the distribution of tasks on peer nodes running the Internet Application Factory. In this context, it implements dynamic load balancing across distinct peer nodes and introduces an overlay network communication system for bridging network boundaries, e.g. allowing peers behind firewalls to share their resources.

5.1.3 Operation

The ODIX implementation is built on top of the XDK and can be basically run on any Java-capable computing device. Before the actual use, however, a suitable ODIX environment has to be deployed and setup, as illustrated in the basic deployment scheme shown in Figure 5.3.

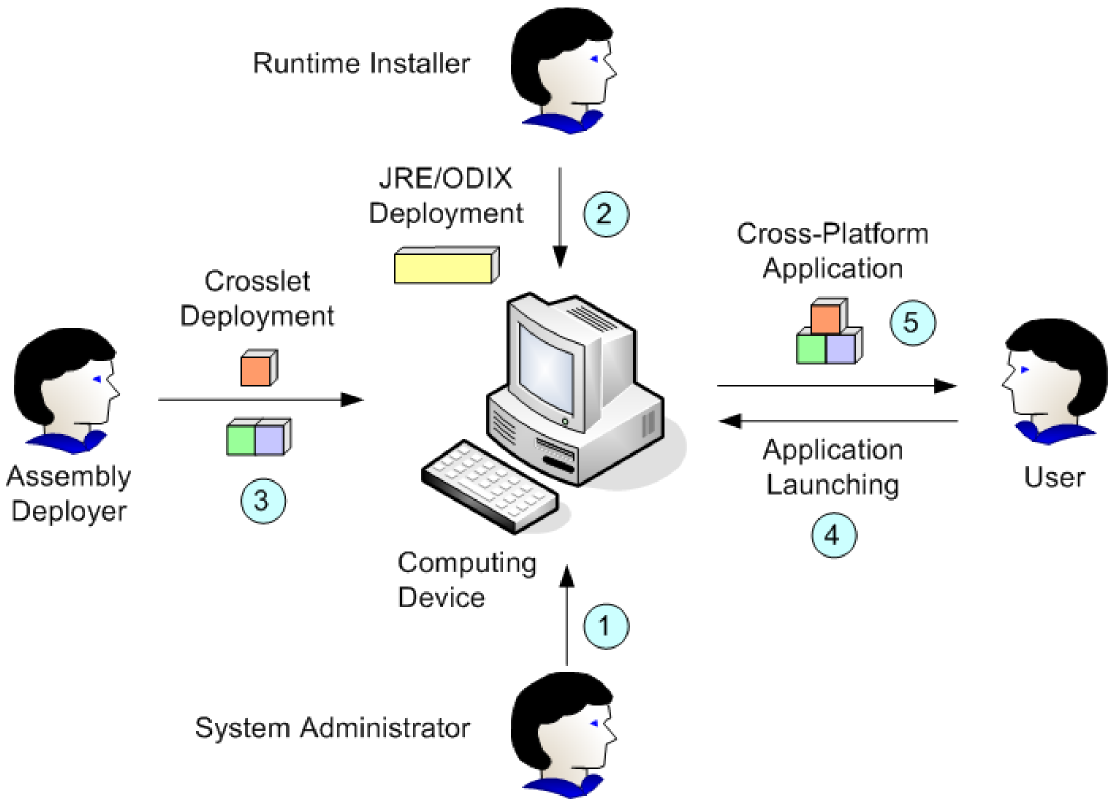


Figure 5.3: Basic Deployment Scheme

Following the user role definitions in Section 2.3.3, there is a system administrator, a runtime installer, an assembly deployer and a user. First, the *system administrator* installs and configures an appropriate operating system to fit the built-in hardware components of the computing device (1). This step is usually performed during the basic device setup and is not actually part of the ODIX deployment process. Next, the *runtime installer* installs a suitable Java Runtime Environment (JRE) and deploys the ODIX core components, e.g. extracting a self-contained ODIX program archive and linking scripts for launching ODIX (2). Besides common ODIX applications for administering ODIX, there are no particular applications deployed so far. This is performed by the *assembly deployer* who copies all potentially required crosslets on the computing device (3). Actually, the crosslets are not immediately installed but simply stored into a directory that acts as a simple file system based crosslet repository. Later on, when ODIX is launched and the *user* attempts to start a not yet installed ODIX application (4), ODIX can determine the required crosslets and retrieve them from the local crosslet repository for assembling the ODIX application (5). Depending on the usage scenario, there are various specialized deployment schemes conceivable, as presented below.

Stationary Deployment. An administrator prepares ODIX to run on dedicated stationary computing devices, e.g. for users in a *personal computing scenario*, as shown in Figure 5.4.

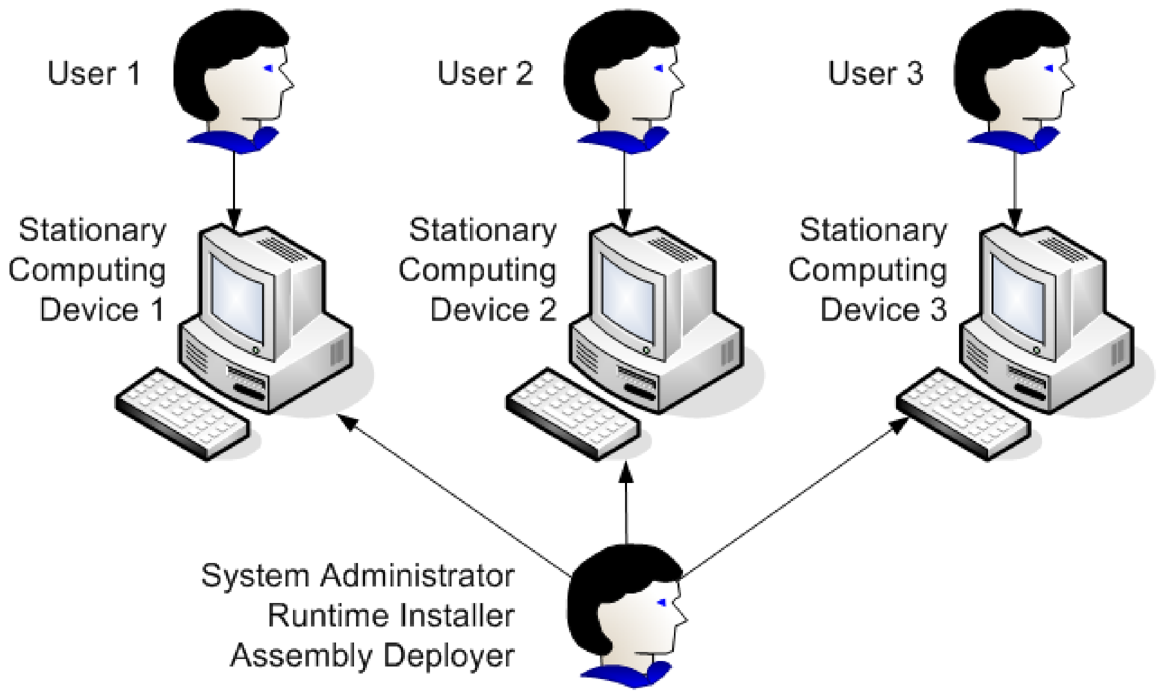

User 1
User 2
User 3
Stationary Computing Device 1
Stationary Computing Device 2
Stationary Computing Device 3
System Administrator
Runtime Installer
Assembly Deployer

Figure 5.4: Stationary Deployment of ODIX

The setup procedure is similar to a regular program installation and is typically performed by manually retrieving the self-contained ODIX program archive, e.g. from the Internet or DVD, and extracting it into a local directory of the computing device. As a result, every user has a separate installation of ODIX and there is no need of an Internet connection unless the user wants to retrieve new crosslets from an Internet crosslet repository. Usually, the same person will complete the tasks of the system administrator, runtime installer and assembly deployer. This deployment scheme also represents an out-of-the-box application of ODIX since no particular network infrastructure or server installations are needed. On the other side, administering a large number of standalone computing devices gets tedious and in an enterprise environment, a system administrator will likely set up a main crosslet repository on a department server and configure the ODIX installations to retrieve updates from there.

Portable Deployment. Another deployment scheme is the *portable installation* which focuses on setting up an ODIX installation on a portable file system that can be alternately attached to *different computing devices*, e.g. for supporting nomadic employees in an *enterprise computing scenario*, as shown in Figure 5.5.

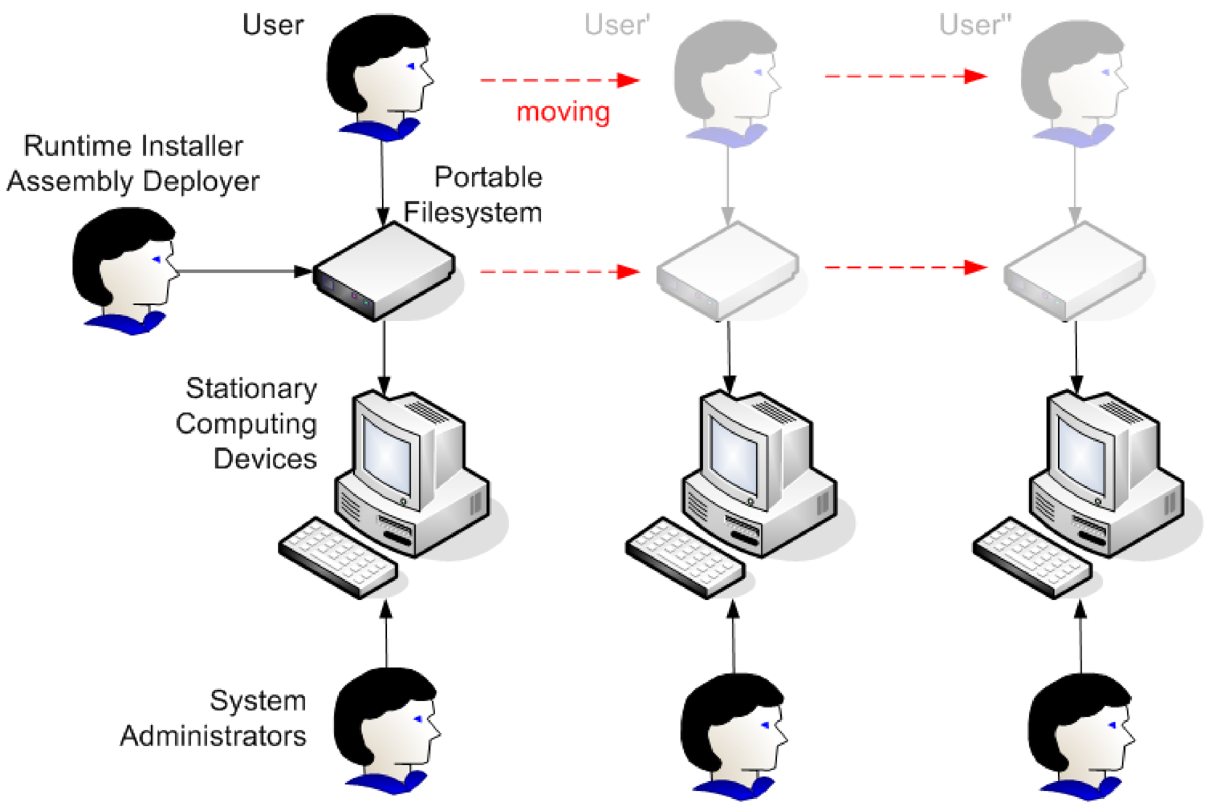


Figure 5.5: Portable Deployment of ODIX

The idea is to decouple the ODIX installation from the running computing device and to allow the launching of ODIX directly from the portable file system, e.g. a USB stick. A user may then move from one computing device to another and launch the same ODIX environment everywhere similarly. In contrast to the stationary deployment, the user settings and installed crosslets are stored on the portable file system that is moved as well. This is similar to the SVM approach in *IoMega v.Clone* that captures and transfers the virtualized application environment of the user by using a mobile disk [174]. As a result, the user takes the complete ODIX environment with him or her. This especially facilitates the usage in untrusted computing environments like in an Internet Cafe or public library. In this context, a particular variant of this deployment scheme is to put several JRE installations on the stick to meet potentially used computing device requirements, e.g. supporting a Linux or a MS Windows environment likewise. This would release system administrators to maintain distinct JRE installations on every computing device and similarly allows users to move to unknown computing systems without assuming the proper installation of a JRE.

Roaming Deployment. The stationary and portable installation schemes are suitable for a small number of computing devices and users, e.g. which are supported by administrators in an enterprise environment. The *roaming use* scheme allows virtually an unlimited number of users to launch ODIX on any Internet computing devices with an appropriate JRE installed, e.g. for running ODIX in a *public computing scenario*, as shown in Figure 5.6.

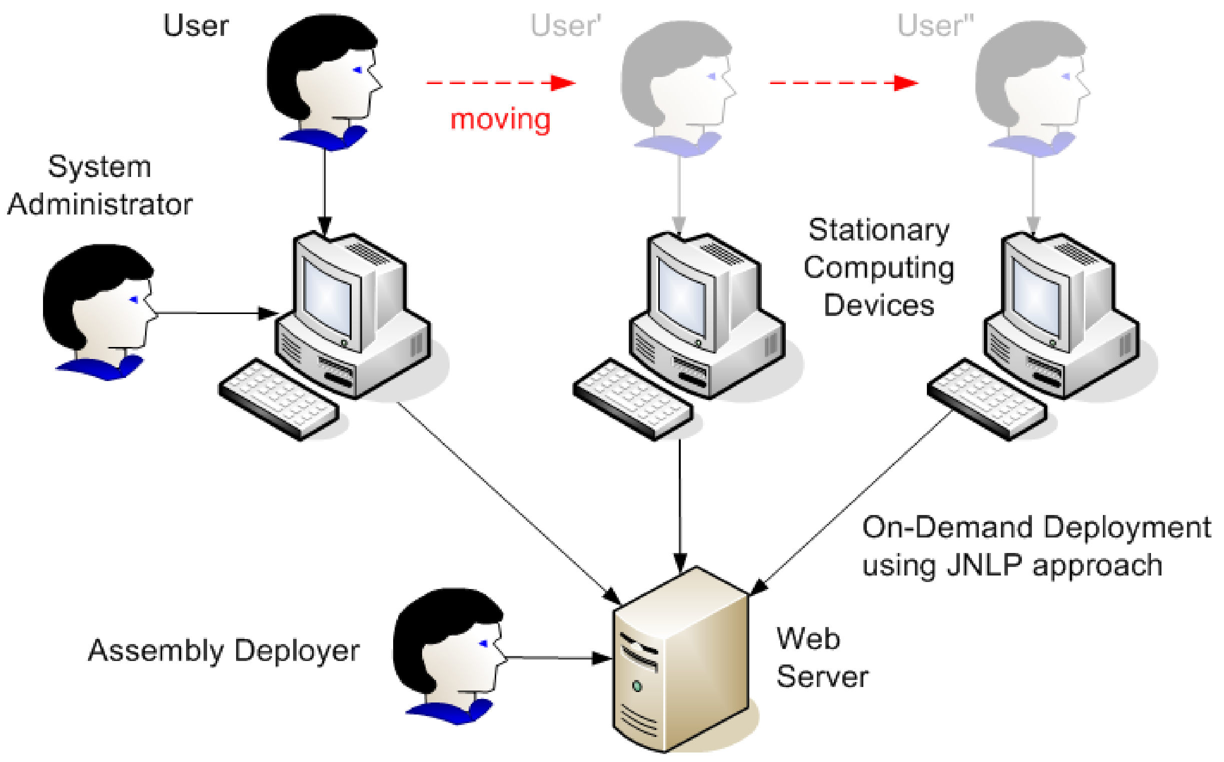

User
User'
User''
moving
System Administrator
Stationary Computing Devices
On-Demand Deployment using JNLP approach
Assembly Deployer
Web Server

Figure 5.6: Roaming Deployment of ODIX

First, an assembly deployer puts the ODIX code on a web server and configures a corresponding Java Network Launch Protocol (JNLP) configuration file. Next, additional crosslets that may be requested by the users are deployed on the web server as well. In contrast to the stationary and portable deployment scheme, there is no particular runtime installer since this task is performed by the JNLP implementation, usually Sun Java Web Start. As a result, if the user wants to launch ODIX, he or she directs the local JRE installation to download the JNLP configuration file from the web server that activates the local Java Web Start installation that starts ODIX. This is usually done by using a regular web browser and visiting a web page on the web server. After ODIX is started, it takes over control and may retrieve the user profile from a local profile directory or from a remote profile server to customize the application environment of the user. In the latter case, ODIX supports the nomadic restoration of the personal settings by using roaming user profiles [see Section 4.5.2]. As a result, a user that has logged into ODIX on one computing device will encounter the same environment on the next computing device he or she logs in.

5.2 On-Demand Application Engine

In this section, the *On-Demand Application Engine* is presented and the use case, features and implementation are described.

5.2.1 Use Case

The overall aim is to support *on-demand task processing* by dynamically selecting, deploying and running appropriate applications on heterogeneous computing resources yet unknown when the computing system is set up, as illustrated in Figure 5.7.

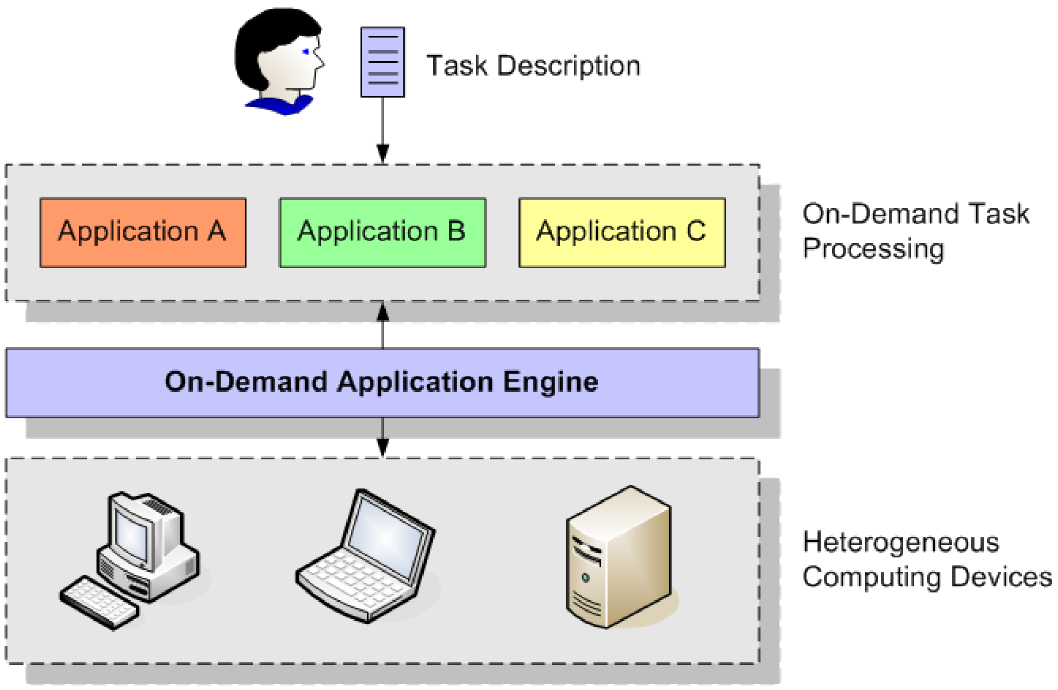

Task Description
Application A
Application B
Application C
On-Demand Task Processing
On-Demand Application Engine
Heterogeneous Computing Devices

Figure 5.7: On-Demand Task Processing

The On-Demand Application Engine is installed in advance on an Internet computing system and represents the basic operating environment to run applications on-demand. It receives task descriptions and launches appropriate applications to perform the requested processing in a self-managing way. The engine entails a simple command-line shell interface that can be used with non-graphical terminals to administer the system installation, to issue task processing requests and to monitor the application execution. Moreover, the engine represents the common runtime foundation for the more specialized implementations of ODIX like the Internet Application Workbench [see Section 5.3] and the Internet Application Factory [see Section 5.4]. In effect, the *On-Demand Application Engine* transforms the employed computing devices into uniform task-processing engines. Custom task processing is enabled by performing the required application deployment in the background and dealing with heterogeneous computing resources without user intervention.

5.2.2 Features

From the developer's point of view, the On-Demand Application Engine inherits the low-level features of the XDK concerning deployment, composition, hosting, customization, interconnection and migration of Internet applications. From the user's point of view, the high-level features of the application engine are as follows.

On-Demand Operating Environment. By installing the application engine, a Java-based operating environment is established on top of heterogeneous computing systems. A user can manage and launch applications in the same way on different nodes on-demand without being concerned about the application deployment and configuration. In this context, a *command-line shell* offers to control the engine and execute custom application scripts, e.g. while installing a software package.

Seamless System Integration. The application engine does not replace the regular operating system but extends it with new functions by using various *system hooks*. First of all, the specific processing of documents is extended by ODIX using the Java activation framework that lets users seamlessly open, edit and print documents with dynamically selected ODIX applications on-demand. In turn, the application engine can launch native applications for processing a given task.

Multi-Tenant and Multi-Tasking Support. In contrast to regular Java runtime environments, the application engine introduces an advanced task management system and allows multiple users to process different tasks at the same time. Moreover, various user sessions may be started and run multiple users applications in the foreground while particular applications can be run as *application services* in the background, e.g. offering access on local computing resources to remote parties.

5.2.3 Implementation

The implementation of the On-Demand Application Engine is based on the XDK framework and adds various components to support the on-demand launching and multi-tasking execution of different application types, as illustrated in Figure 5.8.

The On-Demand Application Engine allows running task executables in distinct runtime environments and with different configurations. For example, if a task can be processed by a native Linux binary and a Java application, the application engine may prefer the native binary on compatible computing platforms and elsewise switch to the Java variant. The active operation mode is determined by evaluating the local system settings and the profile configurations. The system settings contain configurations for initializing the XDK and the application engine. The profile configurations of the user, the computing platform and the applications are stored in the profile registry and used to deduce the self-managing behaviour when processing a given task, as described in Section 4.5.2.

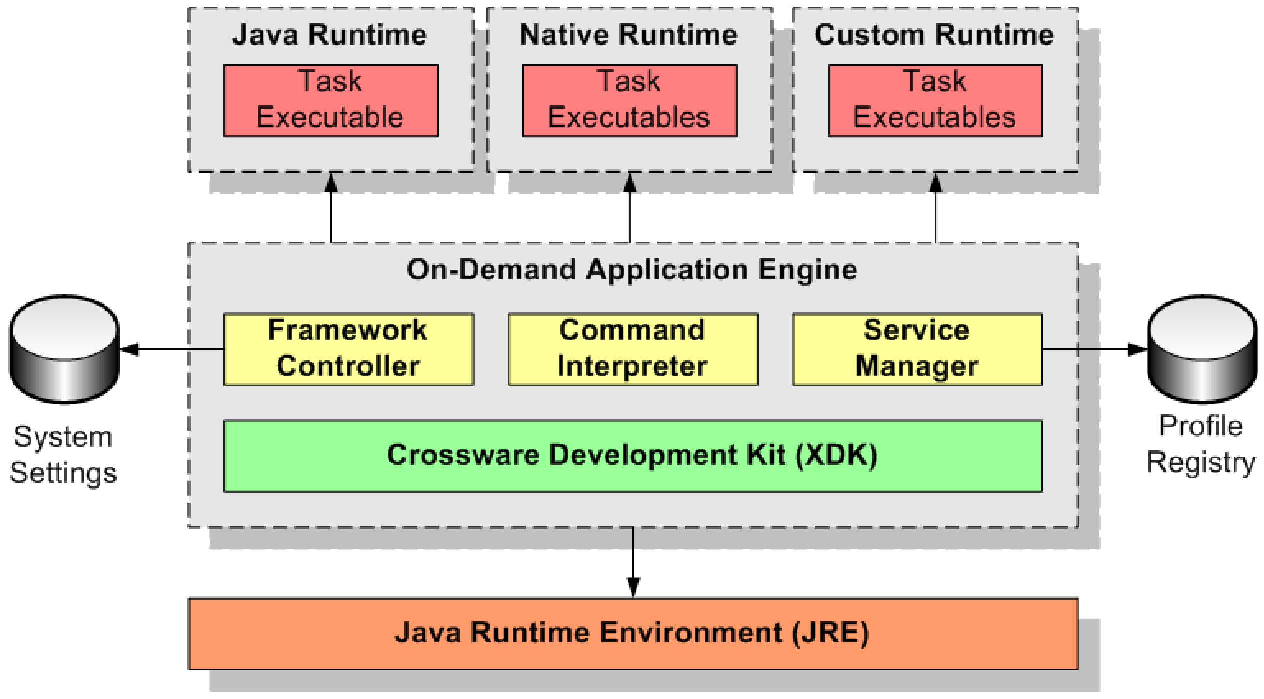


Figure 5.8: On-Demand Application Engine

Framework Controller. The XDK is actually an extensible application framework that is not intended to run solely but which is embedded and initialized by a specific application system. As part of the On-Demand Application Engine, a framework controller is added on top of the XDK that starts its initialization and reads in particular system settings, e.g. scripts for automatically launching system- and user-specific applications at engine start.

Command Interpreter. Besides extending the On-Demand Application Engine dynamically by loading software components during runtime, a command interpreter allows running user-customizable scripts, e.g. for starting runtime services one-by-one or for executing distinct processing statements defined in a task workflow script. A particular objective of the command interpreter is the provision of an interactive command-line shell to the user.

Service Manager. The On-Demand Application Engine primarily executes task-related applications that are launched after having received a task description and that are stopped when the task is completed. In addition, common services are run in the background to support the actual task processing applications, e.g. a peer lookup service or task migration service. To this end, a service manager is introduced that launches and controls regular applications as background services.

5.2.4 Application

The On-Demand Application Engine is not supposed to be used directly by regular users but to launch specific ODIX applications on top of it. From this point of view, a very basic application is the ODIX application console that can be used by administrators to manage an ODIX installation and its resources, as shown in Figure 5.9.

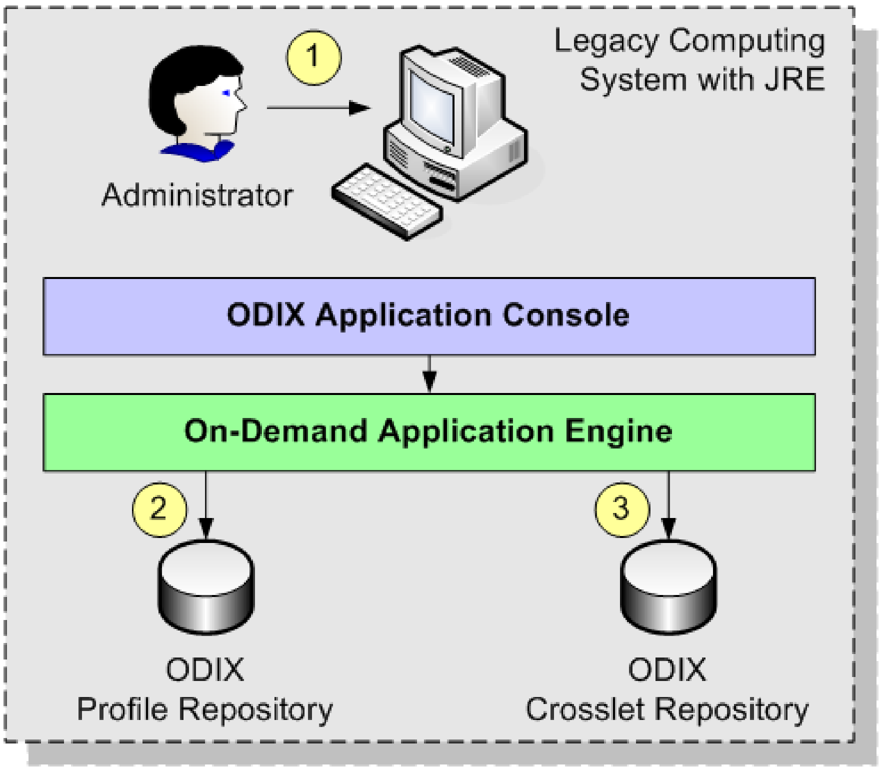


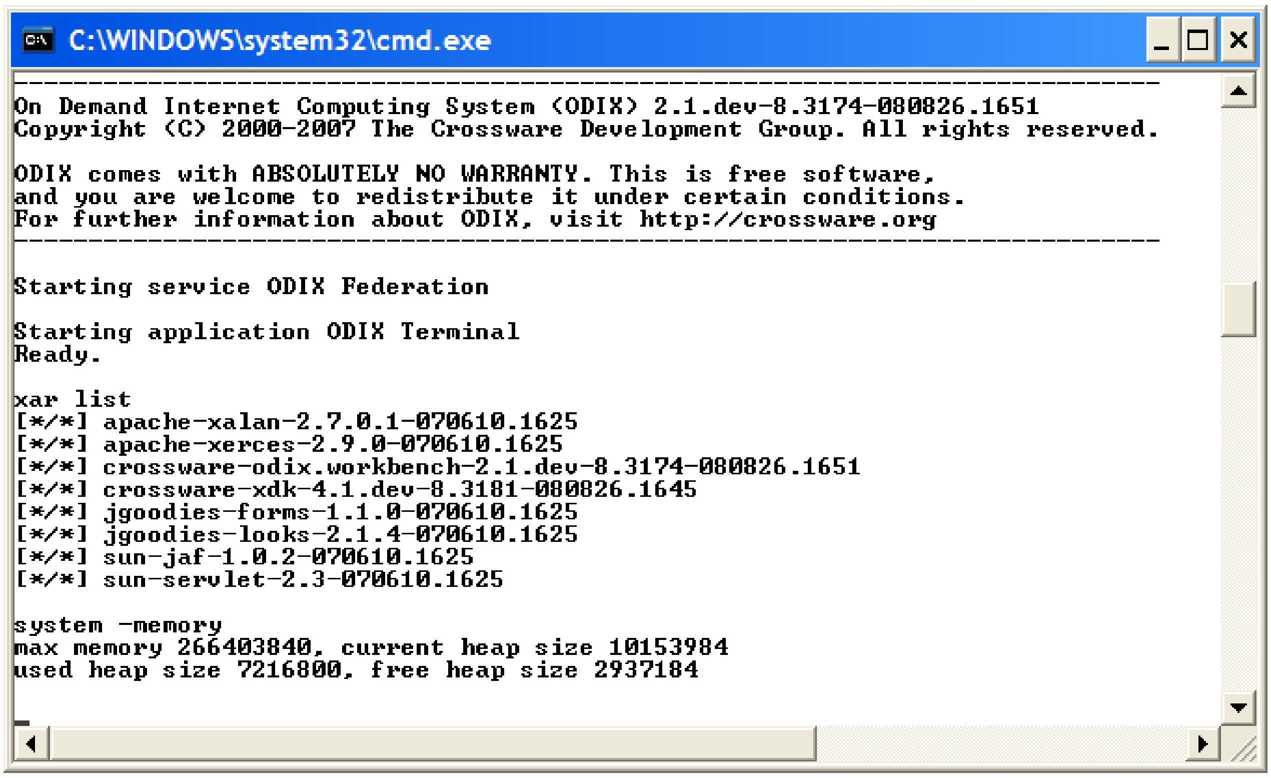
Figure 5.9: Console Administration of the On-Demand Application Engine

The administrator starts ODIX with a particular parameter to launch the ODIX application console (1) which provides scripting access to the local ODIX resources like profile and crosslet repositories (2). At the beginning, the application console executes the *ODIX Autostart Script* from which an exemplary excerpt is shown in Figure 5.10.

```
# start services
service start odix.peer.service
service start odix.router.service
service start xdk.crosslet.update.service

# launch initial applications
launch odix.shell
```

Figure 5.10: Exemplary Excerpt of the ODIX Autostart Script

First, various ODIX services are started in the background, e.g. checking for updated crosslets in the network after the peer discovery service and router service have been started. Then, an interactive shell is opened that allows entering commands and executing further scripts on the command-line, as shown in Figure 5.11.



```
------------------------------------------------------------------------------
On Demand Internet Computing System (ODIX) 2.1.dev-8.3174-080826.1651
Copyright (C) 2000-2007 The Crossware Development Group. All rights reserved.

ODIX comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
For further information about ODIX, visit http://crossware.org
------------------------------------------------------------------------------

Starting service ODIX Federation

Starting application ODIX Terminal
Ready.

xar list
[*/*] apache-xalan-2.7.0.1-070610.1625
[*/*] apache-xerces-2.9.0-070610.1625
[*/*] crossware-odix.workbench-2.1.dev-8.3174-080826.1651
[*/*] crossware-xdk-4.1.dev-8.3181-080826.1645
[*/*] jgoodies-forms-1.1.0-070610.1625
[*/*] jgoodies-looks-2.1.4-070610.1625
[*/*] sun-jaf-1.0.2-070610.1625
[*/*] sun-servlet-2.3-070610.1625

system -memory
max memory 266403840, current heap size 10153984
used heap size 7216800, free heap size 2937184

_
```

Figure 5.11: ODIX Command Line Shell

The screen shot exemplary shows how the user can retrieve a list of all installed crosslets by issueing the command `xar list` and how to check the current memory allocation with `system -memory`. Various commands are available to launch a specific ODIX application or pass a task description that is processed on-demand. The ODIX console also allows administrators to open a shell to remote ODIX installations, e.g. running on server blades without a screen and keyboard device connected.

As a result, the ODIX application engine represents a cross-platform application environment that supports *thread-based multi-tasking* by running Java applications inside a shared JVM or *process-based multi-tasking* by running native applications as regular programs on the operating system level. In conjunction with a crosslet repository infrastructure, it may be used as a particular application deployment and launching tool, e.g. for running Java services in the background and applying software patches on heterogeneous computer systems.

5.3 Internet Application Workbench

In this section, the *Internet Application Workbench* is presented and the use case, features, implementation and application are described. Various parts of the work on the workbench approach have been already published and carried on in [274, 275, 279, 281, 283, 284, 382].

5.3.1 Use Case

The aim of the Internet Application Workbench is to enable *local task processing on-demand* [see Section 2.5.1] by providing a pervasive computing environment across distinct Internet computing devices, as shown in Figure 5.12.

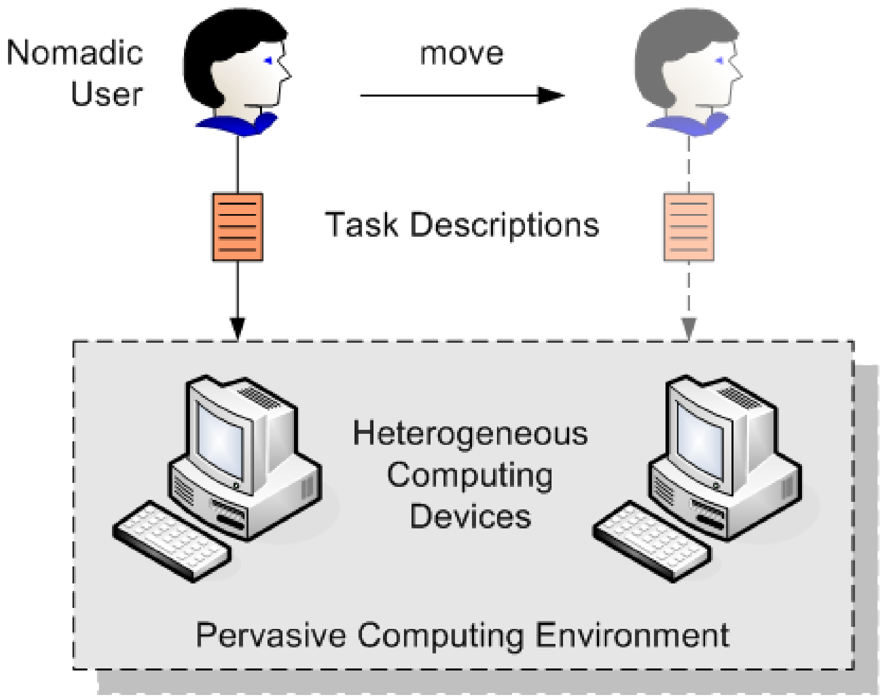

Nomadic User
move
Task Descriptions
Heterogeneous Computing Devices
Pervasive Computing Environment

Figure 5.12: Local Task Processing On-Demand

A user can instantaneously employ a computing device to process his or her tasks without manually installing related applications in advance, e.g. simply by logging into the system and getting seamless access to his or her application preferences and document files. While in use, the device is typically not shared but exclusively used, e.g. a desktop computing system at home. However, public computing resources like downloaded code assemblies may be shared and reused by other users the next time they log in. A related application scenario is *nomadic computing* in which users may move from one computing device to another (personal mobility). From there, a particular need emerges to customize each computing device according to the preferences of the current user, e.g. applying a personalized desktop layout scheme. By using the *Internet Application Workbench*, a user can engage various computing devices to process his or her tasks while having the illusion of a pervasive desktop computing system [274].

5.3.2 Features

From the user's point of view, the Internet Application Workbench represents a graphical user interface to launch applications on desktop computing systems in a similar way as in well-known desktop environments, e.g. MS Windows or Linux KDE. From the developer's point of view, the workbench eases the deployment of desktop applications in a cross-platform operating environment, such as a heterogeneous enterprise network or the Internet. The major features are as follows.

Graphical User Interface. The workbench provides a GUI that can be customized in many ways. First, the workbench can be run in *desktop-mode* providing a fully-featured desktop user interface, e.g. with task bar and window manager. Second, the workbench can be run in *sidebar-mode* and transparently integrating application windows into the native desktop interface. And third, the workbench may be also run in *application-mode* hosting all application windows as internal frames.

Instantaneous Operation. The Internet Application Workbench has not to be installed and set up in advance but it can be dynamically deployed and run instanteneously on any Java-enabled computing system. The core components of the application workbench are packaged into a single assembly that can be easily distributed, e.g. for running the workbench directly from a portable USB stick or for retrieving a single executable file from the Internet.

Pervasive Computing Environment. Another major feature is the synchronization of user settings and documents across distinct desktop computing system. In particular, the workbench allows users to configure an application on one computing system and to restore the application settings on another computing system, e.g. by dynamically retrieving suitable code components and applying related settings in the synchronized user profile.

5.3.3 Implementation

The implementation of the Internet Application Workbench is based on the On-Demand Application Engine (ODAE) and adds the ability to issue task processing requests using a graphical desktop interface, as illustrated in Figure 5.13.

The workbench and its graphical user interface are implemented in pure Java using standard Java Swing components that are part of a standard JRE installation. Due to this, there is no dependency to the locally installed native GUI libraries, e.g. from MS Windows or Linux KDE. And the look-and-feel (L&F) of the workbench is uniform across heterogeneous computing devices [see screen shot in Section 5.3.4].

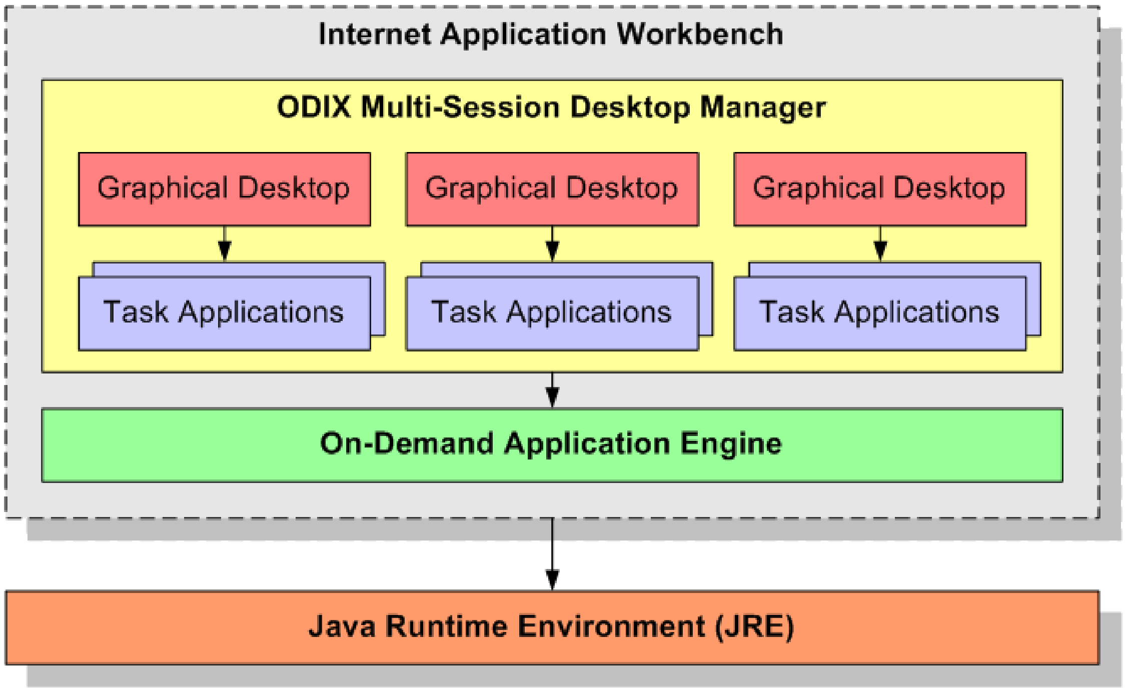

Internet Application Workbench
ODIX Multi-Session Desktop Manager
Graphical Desktop
Graphical Desktop
Graphical Desktop
Task Applications
Task Applications
Task Applications
On-Demand Application Engine
Java Runtime Environment (JRE)

Figure 5.13: Internet Application Workbench

Multi-Session Desktop Manager. The ODIX multi-session desktop manager is a regular ODIX application and launched immediately after the initialization of the ODAE. It is able to manage distinct graphical desktops and application windows on the same screen device by using the multi-session support of the XDK. To this end, a particular window manager API is introduced that has to be used by ODIX application developer to benefit from the ODIX desktop management.

Task Applications. While the application workbench offers users to launch custom *user applications* from a personalized application menu, the workbench also utilizes the application execution engine to determine a suitable *task application* for a given task description. By using the standardized *Java Activation Framework (JAF)* approach, the task handling of the application workbench interacts seamlessly with the underlying legacy operating system [see Section 4.5.1].

Remote Profile Registry. To synchronize the user and application settings across heterogeneous computing systems, the application workbench utilizes the XDK roaming profile implementation [see Section 4.5.2]. At startup, the workbench presents a login dialog to authenticate the user and to check for an updated user profile in the remote profile registry. Afterwards, the local user profile is synchronized, if needed, and the workbench is customized accordingly.

5.3.4 Application

The Internet Application Workbench is supposed to support local task processing on-demand and in this context, a related application scenario is *nomadic computing* and the provision of a pervasive application environment on the desktop computer. The environment seamlessly moves with the user as she or he switches to another desktop computer system and expects her or his customized environment without particular intervention [276]. From this point of view, the *roaming deployment scheme* [see Section 5.1.3] using the Java Web Start approach is usually preferred, as shown in Figure 5.14.

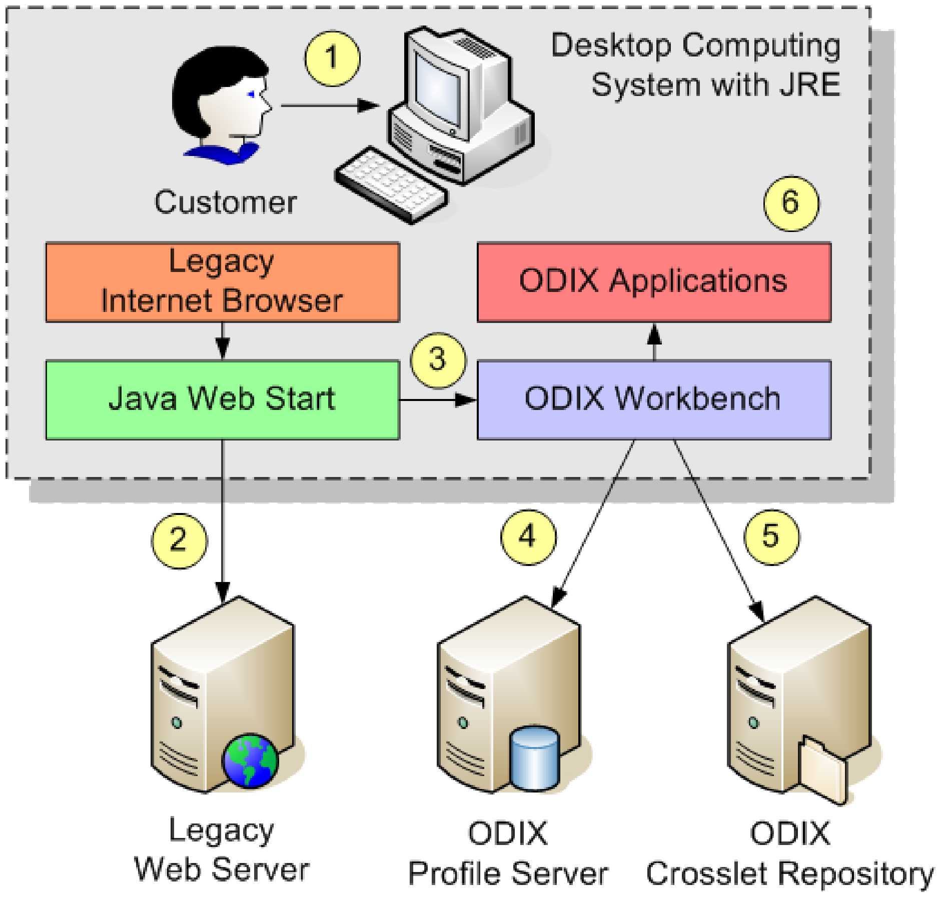
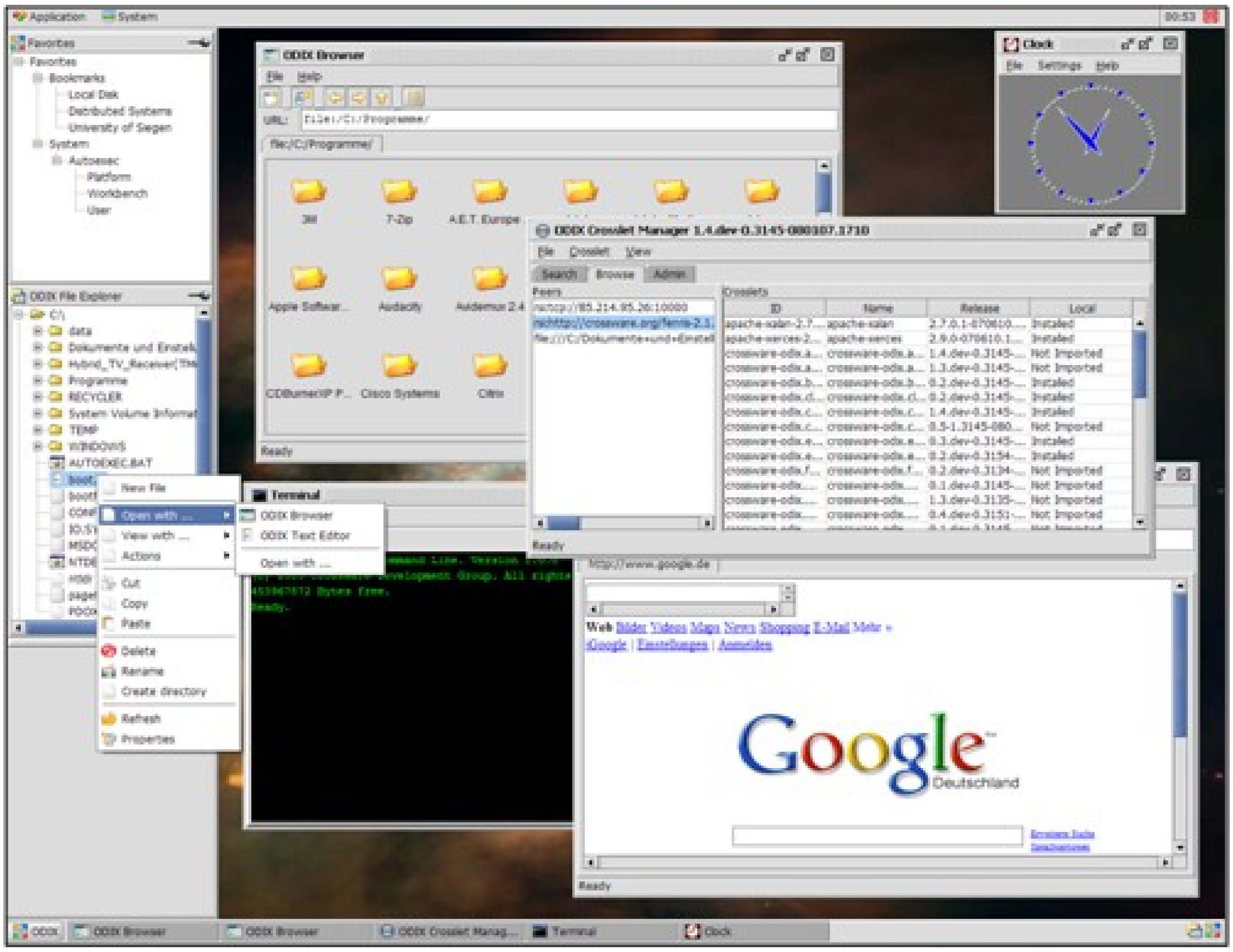


Figure 5.14: Nomadic Computing with the ODIX Application Workbench

A customer uses a legacy Internet browser with Java plugin installed (1) and visits a regular web server with an appropriately configured JNLP file (2). The local Java Web Start installation launches the ODIX workbench on top of the ODAE (3). When the user logs in, the workbench retrieves his or her user profile from a remote profile server and restores the personal settings of the user (4). In this context, the crosslets referred in the user profile are checked and, if needed for the current computing device, retrieved and installed automatically from a remote crosslet repository (5). After all, the workbench opens a graphical user interface, as shown in Figure 5.15.



Peers:
- rnitcp://85.214.95.26:10000
- rnihttp://crossware.org/ferns-2.1.
- file:///C:/Dokumente+und+Einstel

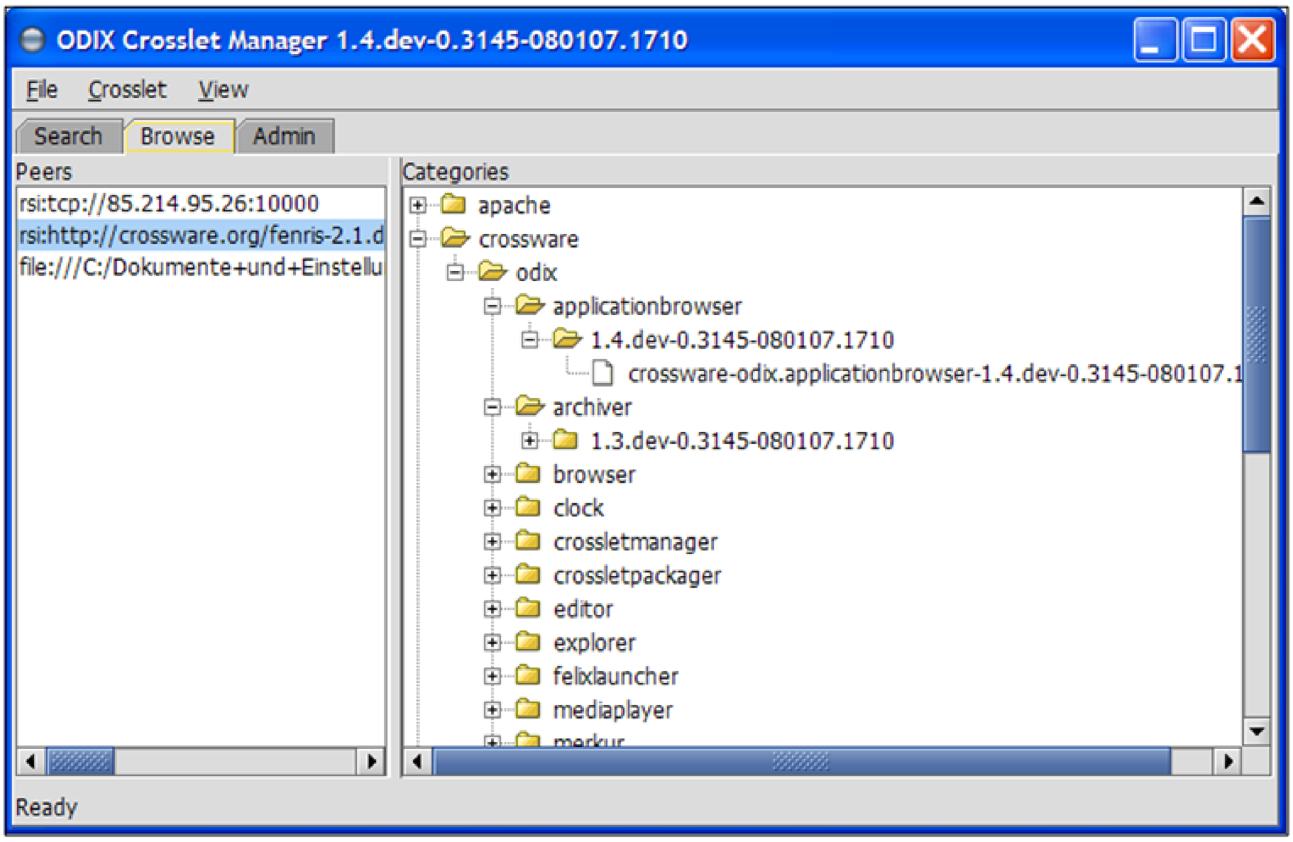
ID	Name	Release	Local
apache-xalan-2.7....	apache-xalan	2.7.0.1-070610.0....	Installed
apache-xerces-2....	apache-xerces	2.9.0-070610.1....	Installed
crossware-odix.a...	crossware-odix.a...	1.4.dev-0.3145-...	Not Imported
crossware-odix.a...	crossware-odix.a...	1.3.dev-0.3145-...	Not Imported
crossware-odix.b...	crossware-odix.b...	0.2.dev-0.3145-...	Installed
crossware-odix.cl...	crossware-odix.cl...	0.2.dev-0.3145-...	Installed
crossware-odix.c...	crossware-odix.c...	1.4.dev-0.3145-...	Installed
crossware-odix.c...	crossware-odix.c...	0.5-1.3145-080....	Not Imported
crossware-odix.e...	crossware-odix.e...	0.3.dev-0.3145-...	Installed
crossware-odix.e...	crossware-odix.e...	0.2.dev-0.3154-...	Installed
crossware-odix.f...	crossware-odix.f...	0.2.dev-0.3134-...	Not Imported
crossware-odix....	crossware-odix....	0.1.dev-0.3145-...	Not Imported
crossware-odix....	crossware-odix....	1.3.dev-0.3135-...	Not Imported
crossware-odix....	crossware-odix....	0.4.dev-0.3151-...	Not Imported

```
[illegible] Command Line. Version [illegible]
[illegible] Development Group. All rights
453967872 Bytes free.
Ready.
```

Figure 5.15: Desktop User Interface of the ODIX Application Workbench

The screen shot shows the application workbench in *desktop mode* where it manages the ODIX task windows in a separate desktop background and covers the backing GUI. The desktop interface is composed of a *taskbar* at the bottom similar to MS Windows or Linux KDE, a *navigation pane* on the left side and a *working pane* on the right side. A user can customize the Look&Feel and store the personal preferences into his or her user profile. The next time the workbench is started, e.g. on another computing device, the user preferences are loaded after login and the interface is restored to the last configuration. Another operation mode of the application workbench is the *sidebar mode* and the seamless integration in the backing GUI, e.g. MS Windows. Then, only the ODIX sidebar, on the left in Figure 5.15, is displayed and can be extended with *widgets*, e.g. an analog clock, a stock market ticker, RSS reader and so on. If an ODIX application is launched, it uses the window manager of the backing GUI and therefore enhances the illusion of the seamless integration in the user's desktop computing environment. Overall, the user can switch to different operating modes and select the integration level of the application workbench into the underlying desktop window system. In conjunction with the cross-platform synchronization of the user profiles, the user gets the illusion of a nomadic desktop environment.

ODIX Software Deployment. A basic ODIX feature is the self-managing deployment and installation synchronization of software components across distinct computing systems by using the XDK Crosslet approach [see Section 4.2.1]. In this context, various ODIX applications allow users to inspect and change the crosslet installations by using a graphical crosslet manager, shown in Figure 5.16.



rsi:tcp://85.214.95.26:10000
rsi:http://crossware.org/fenris-2.1.d
file:///C:/Dokumente+und+Einstellu

apache
crossware
odix
applicationbrowser
1.4.dev-0.3145-080107.1710
crossware-odix.applicationbrowser-1.4.dev-0.3145-080107.1
archiver
1.3.dev-0.3145-080107.1710
browser
clock
crossletmanager
crossletpackager
editor
explorer
felixlauncher
mediaplayer
merkur

Figure 5.16: ODIX Crosslet Manager

The user can check and modify the installed crosslets, e.g. by manually searching a suitable crosslet and trigger the installation. Similar to well-known package manager, such as RPM, the crosslet manager resolves missing dependencies and performs the crosslet installation in a self-managing way. With the same application, a developer can upload a new crosslet to any crosslet repository and thus deploy it without knowing the actual target computing system on which the crosslet will be eventually resolved and installed. As a result, the ODIX Application Workbench is a *Rich Client Platform* that supports local task processing on-demand by retrieving suitable applications from the Internet on-the-fly or on user request.

5.4 Internet Application Factory

In this section, the *Internet Application Factory* is presented and the use case, features, realization and application are described. Various parts of the work on the factory approach have been already published in [66, 235, 281, 318].

5.4.1 Use Case

The goal of the Internet Application Factory is to enable *remote task processing on-demand* [see Section 2.5.2] by providing a shared computing environment on dedicated application servers, as shown in Figure 5.17.

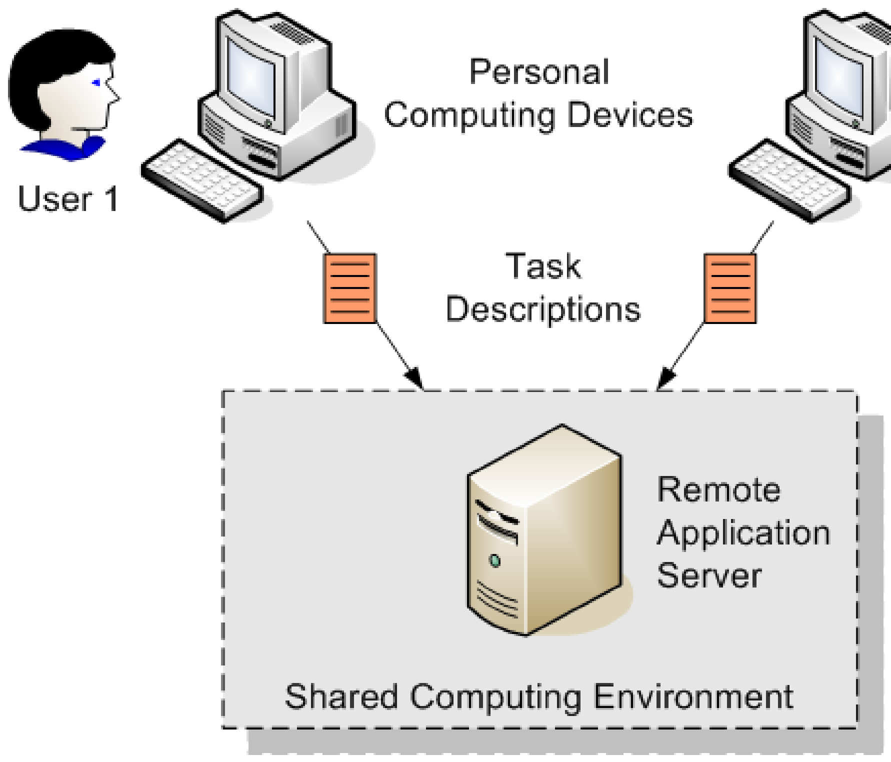

Personal Computing Devices
User 1
User 2
Task Descriptions
Remote Application Server
Shared Computing Environment

Figure 5.17: Remote Task Processing On-Demand

Various users can concurrently utilize one or more remote application server to process their tasks. Typically, a local computing device is used to deploy the task and to retrieve the task results at the end of the processing, e.g. by sending task processing requests via a web browser and downloading the results after task completion. While the access on common computing resources may be shared during runtime, e.g. compatible code components needed to run user applications, others may be protected against mutual access, e.g. session and task data. A particular challenge of the multi-session operation is the concurrent hosting and interaction of user applications that are not known the time the application server is set up. A related application scenario is *utility computing* in which computing resources are shared and utilized on-demand to decrease the overall costs [67]. By using the *Internet Application Factory*, a remote application server can be utilized on-demand to run yet unknown applications for the concurrently processing of various user tasks.

5.4.2 Features

The Internet Application Factory enables the on-demand utilization and sharing of remote computing resources by introducing a self-managing application server. From the user's point of view, the factory is a task processing server that has not to be configured in advance but can be employed like a universal computing resource. From the administrator's point of view, the self-managing factory approach eases the setup of remote computing devices. The major features are as follows.

Shared Resource Utilization. A computing device running the Internet Application Factory can host multiple user sessions and process individual tasks in secured application environments. In contrast to single job assignments, the factory increases the utilization of high-performance computing devices by enqueuing incoming task processing requests. The actual processing may be prioritized according to a given policy configuration, e.g. by favoring gold-level customers.

Remote Task Control. An Internet Application Factory may be installed on any computing device accessible via the Internet. The user interaction is performed by using a remote task control application that enables the controlling and monitoring of the ongoing task processing. After task completion, the task data is written to the given location and can be downloaded afterwards. In addition, a web service interface enables the integration in third-party applications.

Transparent Application Provision. Since an Internet Application Factory is designed to process yet unknown tasks on-demand, there is no way to deploy required applications in advance. Instead, a task processing engine is installed that receives the tasks, checks locally available applications and retrieves missing software components. The actual application provision is performed in a transparent manner, e.g. by determining and reusing suitable applications installed and run before.

5.4.3 Implementation

Like the Internet Application Workbench [see Section 5.3], the implementation of the Internet Application Factory is based on the On-Demand Application Engine (ODAE). It introduces a task execution service for handling and controlling remote task processing requests, as shown in Figure 5.18.

In contrast to the application workbench, the application factory does not limit the effective use to single users who have acquired the hosting computing device. Various users may remotely connect to the factory and enqueue new task descriptions without having physical access to the machine. In addition, the factory does not require a screen device and thus it is particularly suited to be deployed and run on server blades such as found in high-performance computing centers.

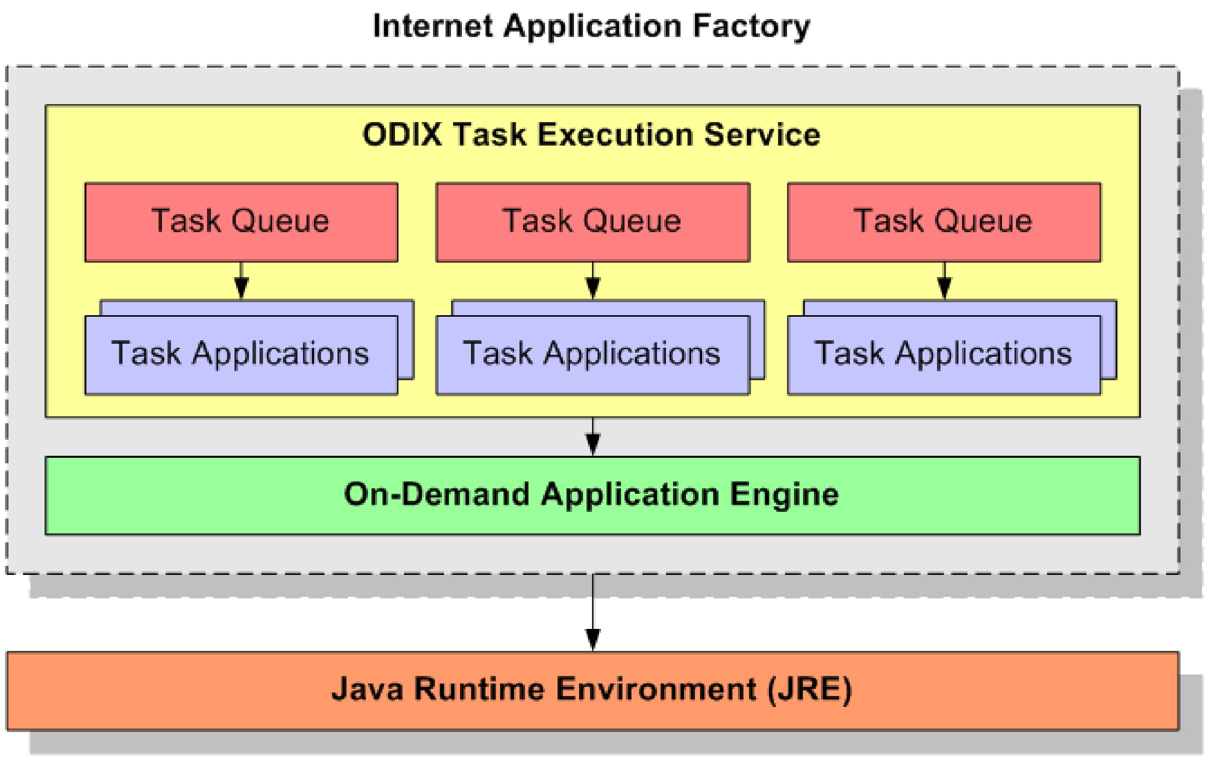

Internet Application Factory
ODIX Task Execution Service
Task Queue
Task Queue
Task Queue
Task Applications
Task Applications
Task Applications
On-Demand Application Engine
Java Runtime Environment (JRE)

Figure 5.18: Internet Application Factory

Task Execution Service. This is the main component of the application factory and handles incoming task descriptions by managing a task queue for every attached user session. It is a regular ODIX service that runs in the background and normally requires no user intervention. Besides a *scheduling interface*, the execution service offers a *monitoring interface* that can be used to track and control the task queues, e.g. via a browser window or a web service connection.

Task Queue. Every user session gets a task queue associated that organizes the incoming user tasks by their priority and competes with other task queues for shared computing resources, e.g. CPU cycles and network bandwidth. A task queue controls the running task applications of the user session, and may suspend and resume task applications to match the current task priorities, e.g. for running low-priority tasks only in the idle time of the task queue.

Adaptive Application Deployment. The task applications needed to process incoming tasks are installed by using the XDK Crosslet approach [see Section 4.2.1]. The crosslet approach helps to reuse local software components and to retrieve task applications that share as many software assemblies with other task applications as possible. The implementation goal is not to retrieve the best application for processing a single task but the ones that utilize the computing system at best.

5.4.4 Application

The basic idea of the application factory is the support of remote task processing on-demand according to the *utility computing* approach [67]. The application factory is launched on a shared computing device that can be remotely accessed and that is able to concurrently process issued task descriptions, as shown in Figure 5.19.

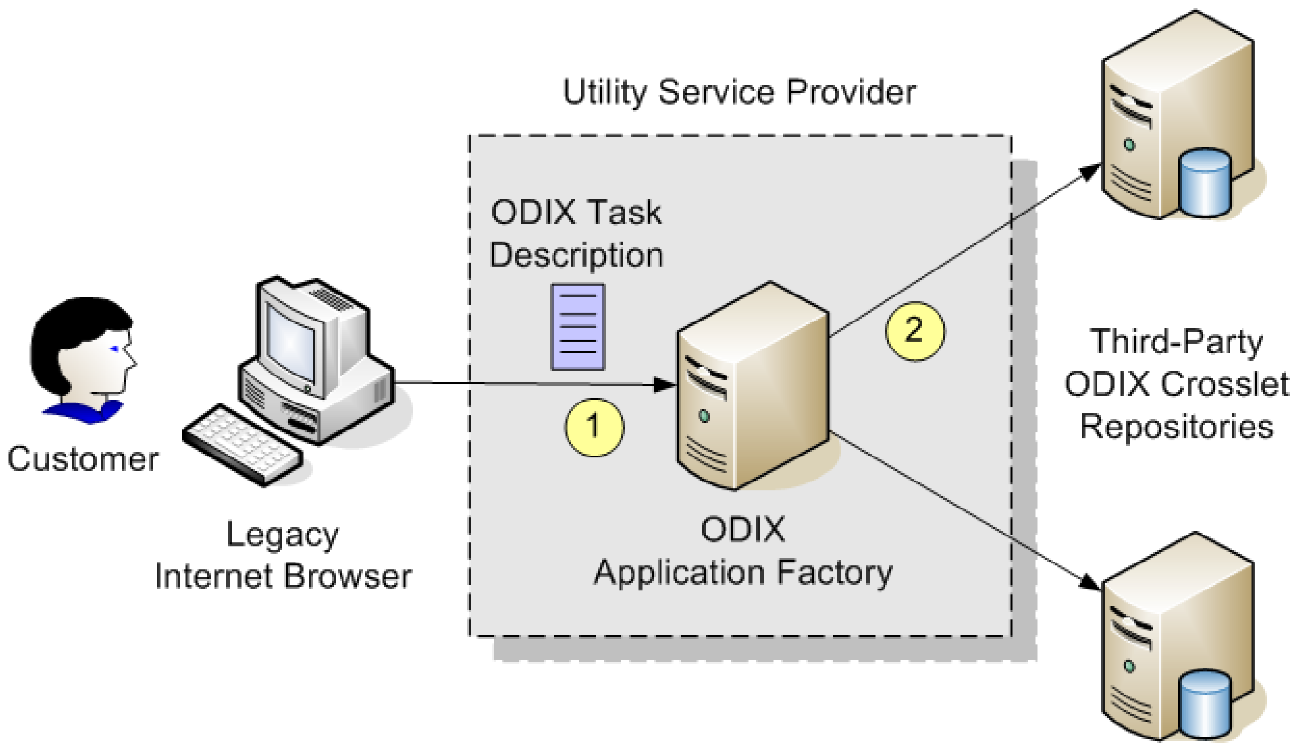
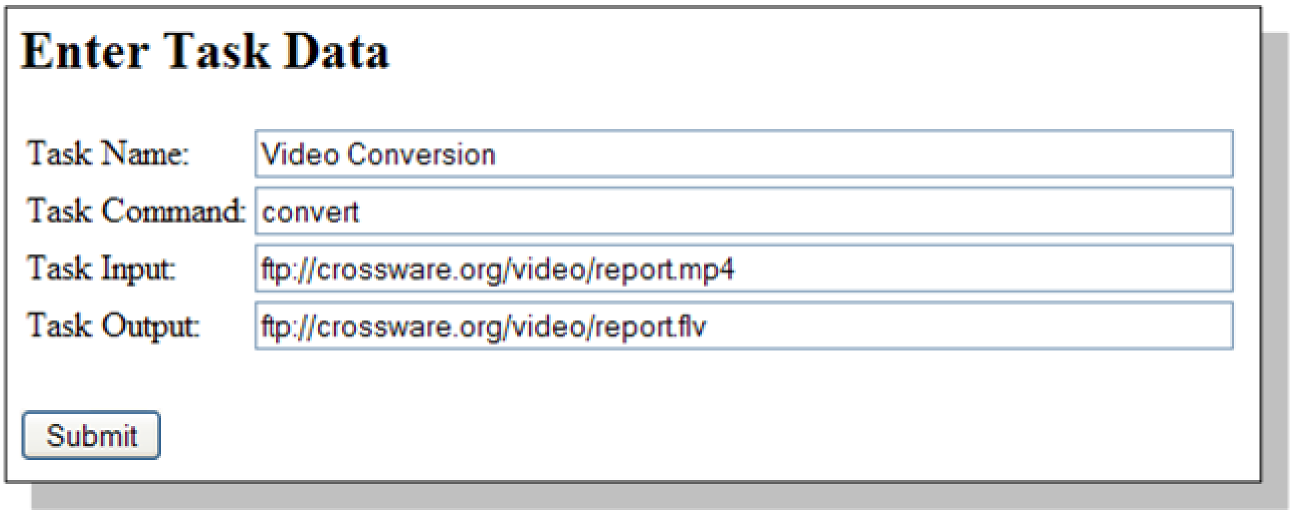


Figure 5.19: Utility Computing with ODIX Application Factory

In a straightforward scenario, a customer may use a legacy Internet browser and visit the task control web page of the application factory. He or she can enter the task data in a web form, e.g. from where to load the task input data, which is then submitted to the ODIX application factory (1), as shown in Figure 5.20.

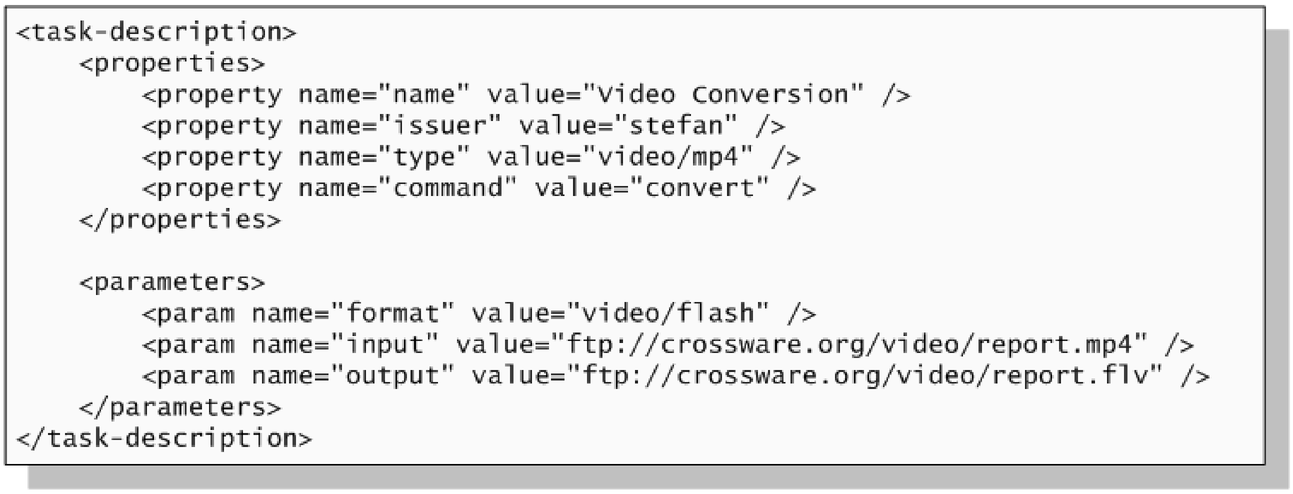


Enter Task Data

Task Name:	Video Conversion
Task Command:	convert
Task Input:	ftp://crossware.org/video/report.mp4
Task Output:	ftp://crossware.org/video/report.flv

Figure 5.20: Simple Task Deployment

A task description is created and enqueued for processing. After an appropriate task application has been determined, required crosslets are downloaded from remote crosslet repositories (2) and installed in a self-managing way. The task is processed and the task output data is written to the given location. The resulting ODIX task description of the example is shown in Figure 5.21.

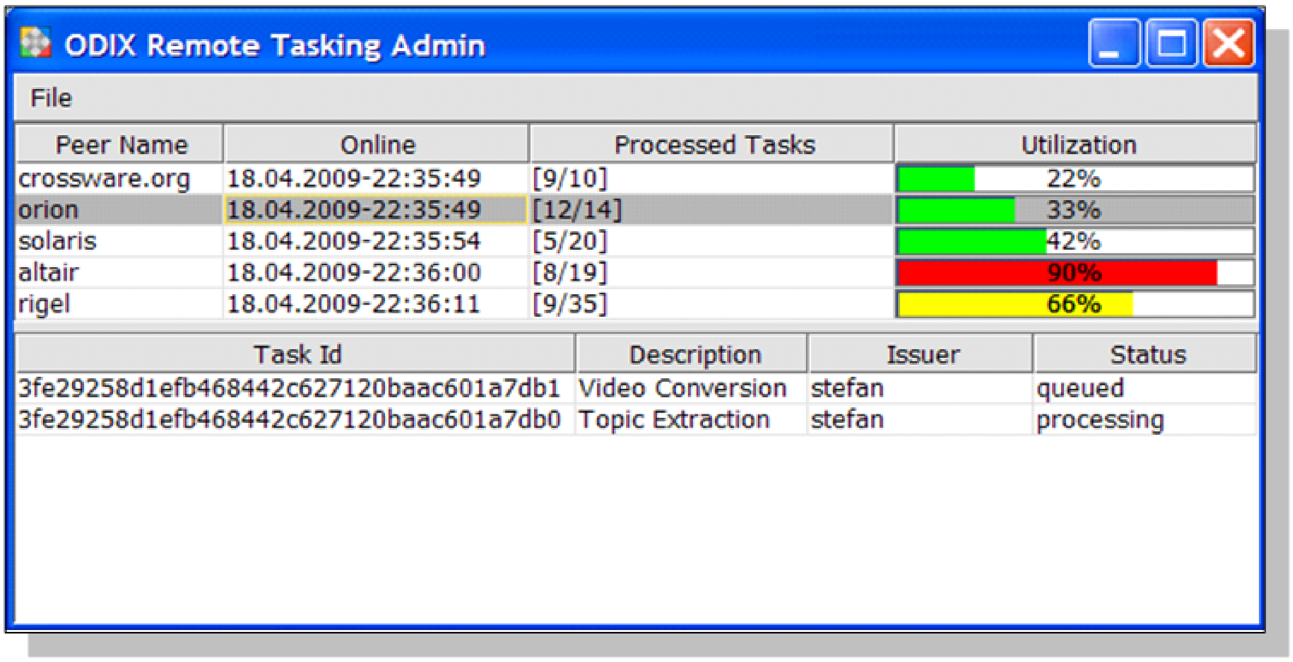


```
<task-description>
    <properties>
        <property name="name" value="Video Conversion" />
        <property name="issuer" value="stefan" />
        <property name="type" value="video/mp4" />
        <property name="command" value="convert" />
    </properties>

    <parameters>
        <param name="format" value="video/flash" />
        <param name="input" value="ftp://crossware.org/video/report.mp4" />
        <param name="output" value="ftp://crossware.org/video/report.flv" />
    </parameters>
</task-description>
```

Figure 5.21: ODIX Task Description

The task description specifies the task properties and parameters that are evaluated by ODIX to compose and launch a suitable task application via the application execution engine [see Section 4.5.1]. The section `properties` contains common task property elements whereas the section `parameters` entails task-specific statements. In the example, the task description requests the conversion of a video stream by using the `type video/mp4` and the `command convert`. In the section `parameters`, the command options `format`, `input` and `output` specifies the conversion of the video to `video/flash` and the locations from where to load the input video and to where write the output video. This simple method is similar to the common program activation scheme found in regular operating systems like MS Windows. It particularly enables ODIX to reuse existing programs, like the GNU tool *convert*, and pass the task data as in the *Common Gateyway Interface (CGI)* approach [368].

Besides using a web browser, a customer may also use the advanced *ODIX Remote Tasking Admin* to deploy tasks and control their processing on an application factory, as shown in Figure 5.22. The application window shows in the upper table the list of known ODIX application factories and the lower table contains the deployed tasks of the selected factory. The customer can inspect the task descriptions and control the execution of his or her tasks, e.g. suspending, resuming and aborting tasks. Further, new application factories can be explicitly added and enqueued tasks can be manually moved from one factory to another.



Peer Name	Online	Processed Tasks	Utilization
crossware.org	18.04.2009-22:35:49	[9/10]	22%
orion	18.04.2009-22:35:49	[12/14]	33%
solaris	18.04.2009-22:35:54	[5/20]	42%
altair	18.04.2009-22:36:00	[8/19]	90%
rigel	18.04.2009-22:36:11	[9/35]	66%

Task Id	Description	Issuer	Status
3fe29258d1efb468442c627120baac601a7db1	Video Conversion	stefan	queued
3fe29258d1efb468442c627120baac601a7db0	Topic Extraction	stefan	processing

Figure 5.22: ODIX Remote Task Admin

Every peer node gets a representative *performance rank*, e.g. by evaluating the CPU type and memory amount or manually set by the node administrator. The *absolute utilization value* of the peer node is dynamically calculated by using the *Java Management Extensions (JMX)* that allows installing advanced runtime monitors in the JVM, e.g. for determining the allocated memory, active thread count, CPU time, number of loaded classes and so on [342]. All measured runtime data are evaluated and normalized to create a *relative utilization value*, as shown as horizontal bars in the upper table in Figure 5.22. The normalization is simply done by dividing the absolute utilization value by the performance rank of the peer node, i.e. peer nodes with a higher performance rank will be favorized among peer nodes with the same relative load.

Finally, application developers may use a web service interface to access the task processing services and integrate them in legacy applications and existing computing infrastructure, e.g. as a service plugin in a media processing system or a service instance in a common *Enterprise Service Bus (ESB)* installation. In fact, the task control web page shown in Figure 5.20 is part of a Java servlet that controls the attached application factory via its web service interface.

5.5 Internet Application Federation

In this section, the *Internet Application Federation* is presented and the use case, features, realization and application are described. Various parts of the work on the federation approach have been already published and carried on in [66, 235, 281, 318].

5.5.1 Use Case

The aim of the Internet Application Federation is to enable *distributed task processing on-demand* [see Section 2.5.3] by providing a federated execution environment in which tasks are dynamically deployed on various computing nodes, as shown in Figure 5.23.

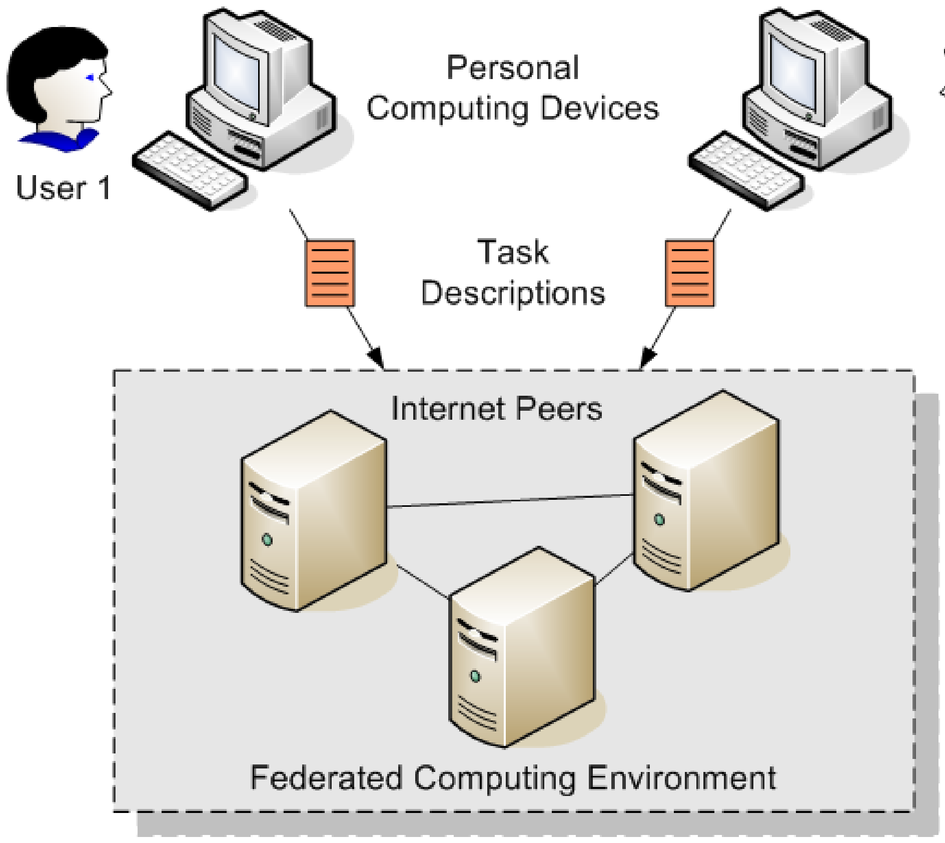


Figure 5.23: Distributed Task Processing On-Demand

In a large distributed computing environment, many computing nodes may be available for processing tasks. This can be used to split and to deploy complex tasks to multiple nodes at the same time for increasing the overall throughput. In addition, the task overloading of single nodes can be avoided by balancing the tasks within a group of computing nodes, e.g. using a task scheduler before task deployment and proactive migration support during ongoing task processing. A related application scenario is *public computing* in which yet unknown Internet computing devices are utilized to process a large number of user tasks. A major challenge is the transparent employment of loosely coupled and separately managed computing systems to run and connect suitable applications, e.g. crossing platform and network boundaries. By using the *Internet Application Federation*, a user can deploy multiple tasks to an Internet peer federation that transparently handles deployment and balancing of user tasks as well as required code distribution to perform the task.

5.5.2 Features

The Internet Peer Federation enables the grouping of distinct Internet computing devices into a self-managing computing federation by using a peer-to-peer interconnection approach. From the user's point of view, the federation acts as a single huge computing system that hides the presence of individual computing nodes and different network connections. The implementation also supports the self-managing migration of task processes across various nodes.

Federated Internet Peers. The grouping of Internet computing devices is performed by using a self-managing super peer organization approach. Selected peers are configured with each other's network address and form an initial super peer network. After starting up the Internet application factory, a regular peer performs a peer discovery and joins the peer federation by connecting to a super peer, e.g. located in the same Intranet and determined by using a multi-cast approach.

Overlay Network Communication. A peer federation is typically spread across various networks with different connection parameters and firewall settings. For peer inter-communication, an overlay network communication approach is used that allows peers to dynamically negotiate a suitable network protocol, e.g. by using the Java method streaming approach. In addition, particular super peers may be used as relay station to allow connecting peers behind a firewall.

Balanced Task Processing. A particular feature of the Internet Peer Federation is the ability to balance the task processing across various peers. A super peer permanently monitors the load of the connected peers and if a new processing request is received, it delegates the task to a peer that is able to process the task. If the overall throughput could be increased by including additional peers, e.g. having joined the federation recently, the super peer may migrate running tasks to another peer.

5.5.3 Implementation

The implementation of the Internet Application Federation is based on the On-Demand Application Engine (ODAE) and introduces various services to enable automatic task distribution across various Internet Application Factories, as shown in Figure 5.24.

The application federation is basically composed of several nodes with different responsibilities. A master node (location A) controls various worker nodes (location B and C) that operate as application factories [see Section 5.4]. The master node is well-known to clients and acts as the task scheduling node that distributes incoming task processing requests to the application factories according to an autonomic load balancing policy.

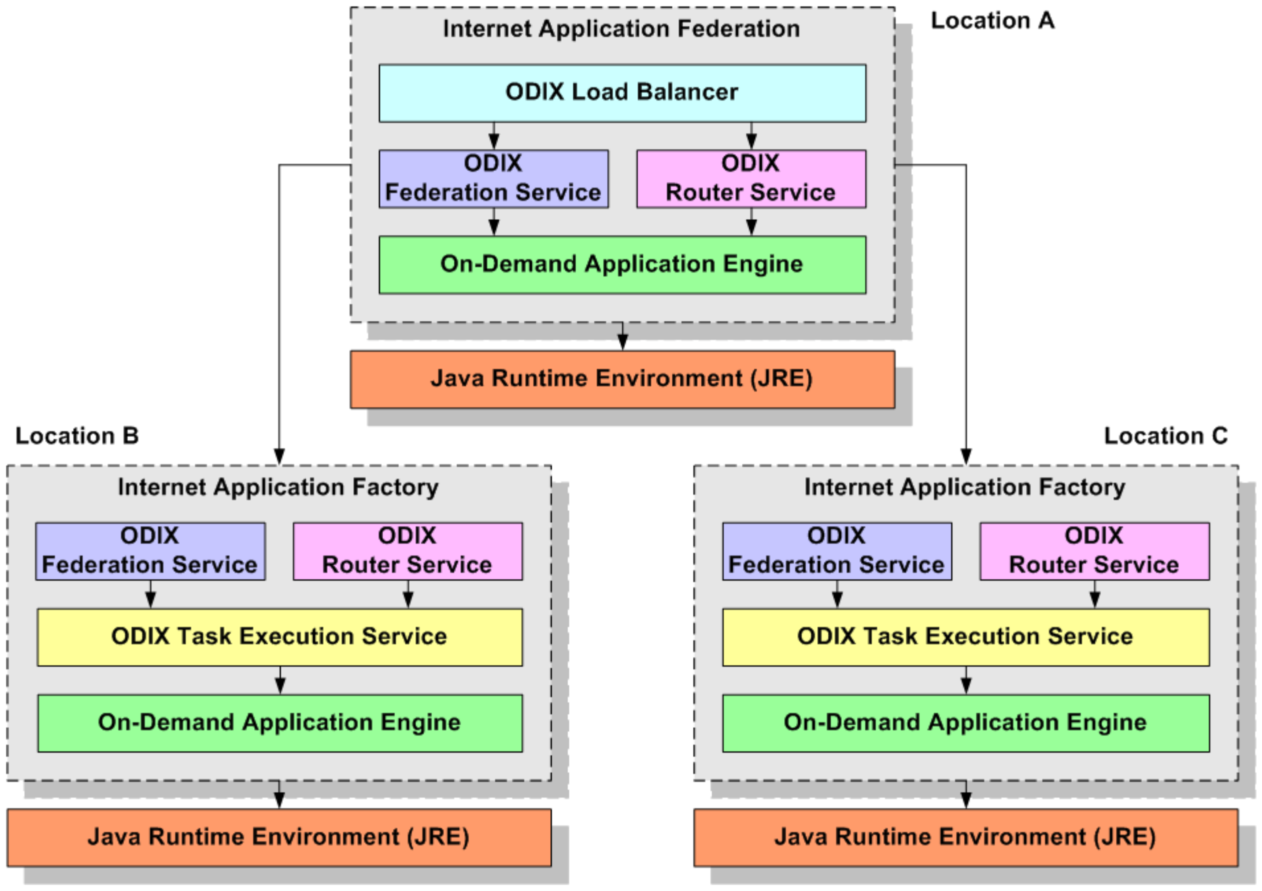

Location A
Internet Application Federation
ODIX Load Balancer
ODIX Federation Service
ODIX Router Service
On-Demand Application Engine
Java Runtime Environment (JRE)
Location B
Internet Application Factory
ODIX Federation Service
ODIX Router Service
ODIX Task Execution Service
On-Demand Application Engine
Java Runtime Environment (JRE)
Location C
Internet Application Factory
ODIX Federation Service
ODIX Router Service
ODIX Task Execution Service
On-Demand Application Engine
Java Runtime Environment (JRE)

Figure 5.24: Internet Application Federation

Federation Service. This service dynamically groups unmanaged peers to a task processing federation. The default implementation establishes a *super peer network* following the *SG-1* approach [245]. Every application factory that is supposed to be included in a federation, installs a corresponding federation service. By replacing the service implementation with another one, different peer grouping approaches can be run, such as a multi-cast packet discovery solution in an Intranet.

Router Service. The service implementation is based on the virtual object communication approach of the XDK [see Section 4.6] and adds the ability to route peer messages over multiple peers, similar to the TCP/IP router approach. The routing uses *Java Remote Method Streams* and may dynamically bridge different network protocols. Once established, an existing communication channel between two peers is shared among distinct applications and in both directions.

Load Balancer. The distribution and migration of running tasks is performed by the load balancer service. In a simple implementation, it tracks all known peers of a federation and it delegates incoming task processing requests to every peer in a round-robin fashion. While monitoring the peers for processing a task, the load balancer service may also migrate single tasks from one peer to another by using the XDK *Ad-Hoc Execution Migration* approach [see Section 4.7].

5.5.4 Application

The application federation is designed to support the *distributed task processing on-demand* as proposed in the public and cloud computing approaches [220]. Various application factories can be manually or dynamically grouped, e.g. by using a super peer network implementation like SG-1 [245], and used to distribute incoming task processing requests on multiple computing devices, as shown in Figure 5.25.

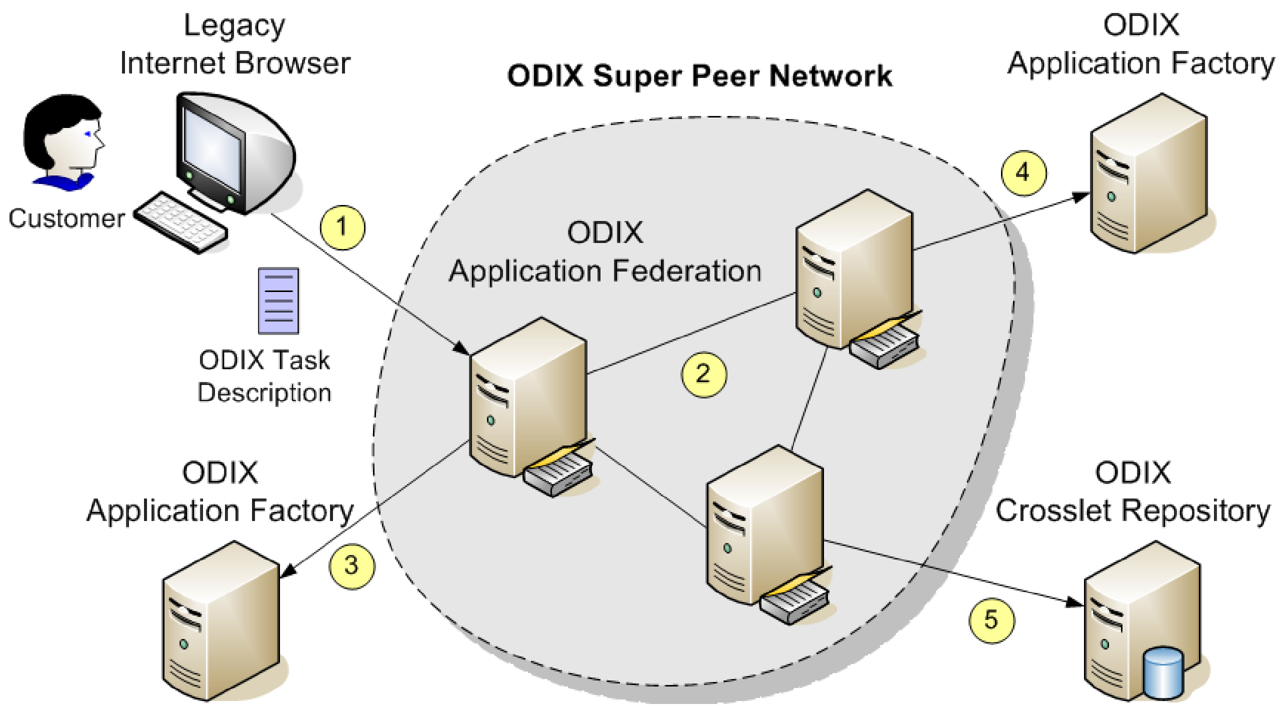
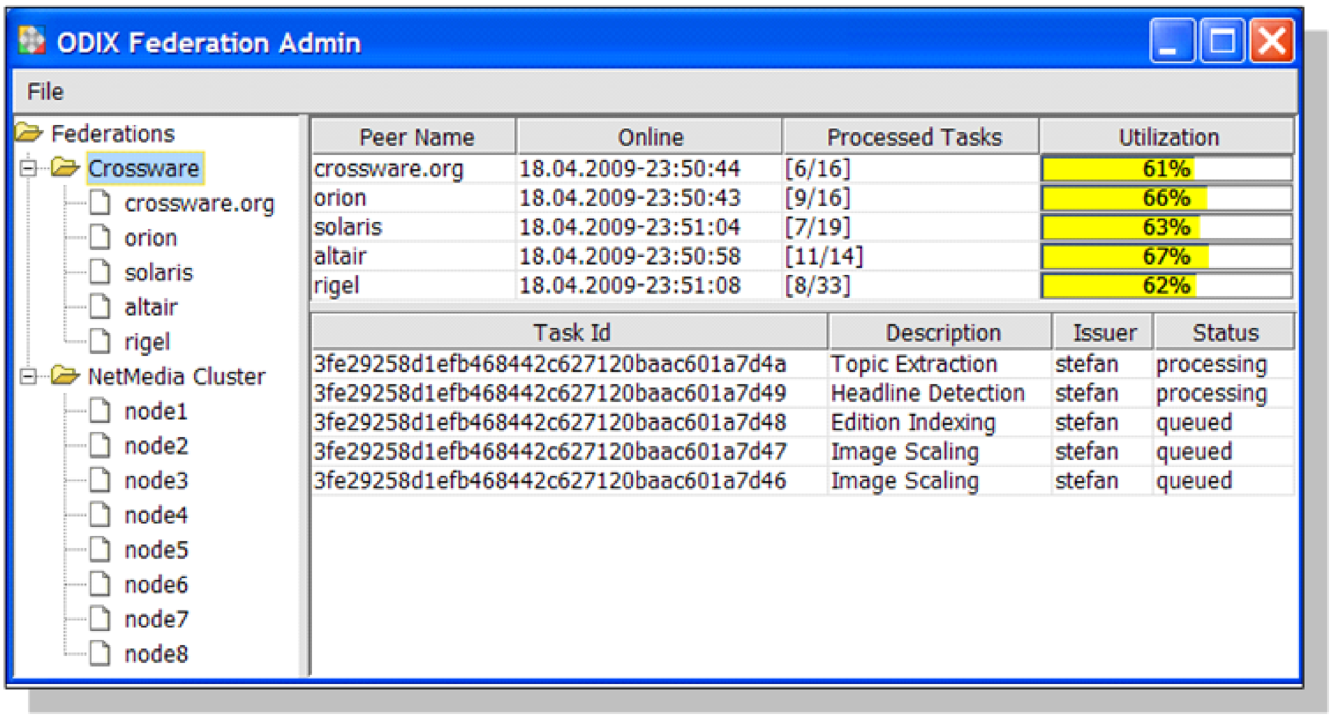


Figure 5.25: Public Computing with the ODIX Application Federation

A customer sends a task description to a master federation node (1) without actually knowing the factories being part of the federation. As with ODIX application factories, this can be easily done by using a regular Internet browser and uploading the task data via a web form [see Section 5.4.4]. Depending on the task processing requirements and load distribution policy, the federation determines one or multiple suitable factories (2) and delegates the task processing (3, 4). The further processing workflow is similar to the one implemented by the ODIX application factories, i.e. retrieving required crosslet components and launching a suitable task processing application (5).

After the task has been processed, the results are returned to the delegating master node and stored until the customer retrieves the processed data, e.g. by visiting the federation control web page and downloading the task output data or following a URL, e.g. pointing to a location on a shared file server in an enterprise network. In addition to the federation control web page, the *ODIX Federation Admin* application allows administrators to inspect the current federation organization and monitor the utilization of every node by using a rich client interface, as shown in Figure 5.26.



- Federations
  - Crossware
    - crossware.org
    - orion
    - solaris
    - altair
    - rigel
  - NetMedia Cluster
    - node1
    - node2
    - node3
    - node4
    - node5
    - node6
    - node7
    - node8

Peer Name	Online	Processed Tasks	Utilization
crossware.org	18.04.2009-23:50:44	[6/16]	61%
orion	18.04.2009-23:50:43	[9/16]	66%
solaris	18.04.2009-23:51:04	[7/19]	63%
altair	18.04.2009-23:50:58	[11/14]	67%
rigel	18.04.2009-23:51:08	[8/33]	62%

Task Id	Description	Issuer	Status
3fe29258d1efb468442c627120baac601a7d4a	Topic Extraction	stefan	processing
3fe29258d1efb468442c627120baac601a7d49	Headline Detection	stefan	processing
3fe29258d1efb468442c627120baac601a7d48	Edition Indexing	stefan	queued
3fe29258d1efb468442c627120baac601a7d47	Image Scaling	stefan	queued
3fe29258d1efb468442c627120baac601a7d46	Image Scaling	stefan	queued

Figure 5.26: ODIX Federation Admin

The left pane shows the discovered peer federations, e.g. managed by super peers or manually grouped by an administrator, the right upper table lists the peers of the selected federation and their properties, e.g. number of processed tasks and their relative utilization [see Section 5.4.4], and the right lower table contains information about the tasks of the selected peer or the entire federation. If a peer node is added to the federation, a peer description is sent to the master node to register the available resources on the peer, as shown in Figure 5.27.

In the excerpt, the peer node provides information about its CPU, memory, installed native operating system, the version of the installed JRE and ODIX. This can be particularly used by the master node to distribute native task applications, e.g. MS Windows applications, on peer nodes capable to run the program executable. Due to varying task processing efforts, task application requirements and available peer resources, it is difficult to calculate the resulting load of a peer node before the task application has been deployed and launched [208, 323].

```xml
<peer-description>
    <properties>
        <property name="name" value="orion" />
        <property name="cpu" value="i686-2600" />
        <property name="memory" value="2048m" />
        <property name="os" value="win32" />
        <property name="java" value="1.5.0_14 1.6.0_13" />
        <property name="odix" value="2.1.0" />
    </properties>
    ...
</peer-description>
```

Figure 5.27: Excerpt of a Federation Peer Description

There are advanced load-balancing approaches for P2P networks [136] and that particularly tackle performance prediction, e.g. by using utility functions to enable autonomic workload mapping in a cloud-based distributed computing system [292]. The ODIX application federation, however, uses a simple self-managing load-balancing approach that first reads in all static performance indices of the available peer nodes and monitors their relative utilization values. If a task processing request is to be deployed by the master node, it selects the peer node with the least relative load. Later on, the master node monitors the current load of all processing peers and may dynamically migrate running task applications from an overloaded peer to another by using the XDK *Ad-hoc Execution Migration* approach. Of course, this is only possible with supported ODIX applications as described in Section 4.7. Further, task applications are only migrated across the peers of the same federation which means that a peer federation as a whole may become overloaded. In this case, distinct task applications may also be temporarily suspended and their task data is written to disk. Finally, if the master node realizes that a peer node is no longer part of the federation, it redeploys the tasks to another node and marks the previous task deployment as invalid.

5.6 Summary

In this chapter, the *On-Demand Computing System (ODIX)* has been described that is designed to support local task processing, remote task processing and distributed task processing in the Internet, as introduced in Chapter 2. It is based on the idea of building an *Autonomic Cross-Platform Operating Environment (ACOE)*, which has been presented in Chapter 3, to enable on-demand task processing in a public computing scenario. By using the *Crossware Development Kit (XDK)* presented in Chapter 4, ODIX benefits from various self-managing features towards deployment and composition, hosting and customization, and interconnection and migration of related Internet applications, as briefly discussed below.

On-Demand Application Engine. The ODIX implementation is based on an *On-Demand Application Engine* that provides the basic functions to launch distinct Internet applications on-demand, to manage concurrently processed tasks and to integrate the overall task processing into the currently engaged computing system. Based on the XDK, ODIX can be run on any Java-capable computing device and supports stationary, portable and roaming deployment depending on the use case. In this context, it should be pointed out that ODIX is an integrated implementation and the set up of a JRE is the only prerequisite. After deploying the engine, all further ODIX components can be retrieved from the Internet while being online. In particular, the application engine automatically installs application packages needed to process an on-demand computing task by evaluating the task description. In comparison to existing solutions like Eclipse Equinox, the application engine is not limited to deploy and run Java applications only. It can evaluate the local operating system, prepare additional application environments and launch custom executables, e.g. by retrieving an optimized native library to run a computational task. From this point of view, ODIX may be used as a generic software installer that supports cross-platform application management in a heterogeneous and spatially distributed environment. However, the proprietary packaging of code assemblies and application description represent the major disadvantages compared to established approaches like Java Web Start and OSGI-based solutions.

Internet Application Workbench. Based on the On-Demand Application Engine, the *Internet Application Workbench* has shown the support for local task processing on-demand by using a graphical user interface. This approach eases the regular user interaction and offers the seamless integration into the desktop computing environment, e.g. by being launched as sidebar application, hosting graphical widgets and opening legacy application windows. From a user's point of view, the application workbench represents a rich client computing system that can be easily deployed using

Java Web Start and a regular Internet browser. In addition to related rich client implementations like *Eclipse RCP*, the application workbench utilizes the XDK roaming profile approach and offers the synchronization of user settings and documents. In a nomadic use context, the application workbench provides the illusion of a pervasive application environment. However, this is only valid for ODIX applications being specifically implemented to be used in the workbench and for third-party Java applications using the regular Java preferences approach. User preferences of native applications that rely on operating system features, such as the MS Windows registry, are not synchronized. From a developer's point of view, the application workbench enables the integrated and collaborating execution of multiple graphical desktop applications in an integrated environment. The seamless integration of application windows, however, is only possible for ODIX applications using the workbench API and the ODIX window management.

Internet Application Factory. Another ODIX component is the *Internet Application Factory* that supports the remote task processing on-demand by offering access to a shared remote computing resource and introducing a task execution service to handle multi-session task processing. For this purpose and in contrast to the application workbench, an administrator has to deploy and set up the application factory manually on a dedicated computing device first, e.g. on a high-performance server. There is no way to deploy the application factory on a remote computing system without administrator intervention. In this context, the administrator will also have to adjust the computing policies for different users, e.g. the task priority and available computing time. A particular feature of the application factory is the ability to manually control the task execution, e.g. to temporarily suspend a task execution. However, this is only possible for ODIX applications that have been implemented using the XDK thread execution control. In this context, security concerns arise from the concurrent execution of multiple task processes in the same JVM which have been not addressed in this thesis. Distinct task applications may influence each other in terms of thread scheduling, memory allocation and resource deadlocks. Further, the performance rank setting and the evaluation of runtime properties to calculate the relative utilization value of the application factory assumes that ODIX is installed on a dedicated computing device. A different approach is needed if the application factory is run on a multi-purpose device, e.g. in the background on a desktop computer system.

Internet Application Federation. The *Internet Application Federation* supports distributed task processing on-demand by connecting multiple application factories and balancing the task deployment. The setup of an application federation is quite simple by installing the ODIX federation service on every participating application factory and initiating the building of a dynamically managed super peer network. By using the XDK remote method streaming implementation, the federation interconnection is not limited to a specific protocol but can negotiate the connection parameters during the peer inclusion phase. Moreover, the ODIX router service let the hosting application factory act as a network router between factories without direct connection, e.g. two machines separated by firewalls. From this point of view, the application federation is a self-managed peer federation that automatically negotiates network protocols and routes. The disadvantage is the proprietary realization that complicates the integration of the ODIX federation implementation in an existing public or cloud computing installation. A distinct feature of the application federation is the ability to balance running task applications between multiple application factories by monitoring the relative load of the peers and using the XDK ad-hoc execution migration implementation. Of course, this is only valid for ODIX applications that support the suspension and streaming of task threads. In this context, there is no protection against data loss due to unexpected server shutdown or crash.

In summary, ODIX represents a pure Java implementation to support on-demand task processing in a heterogeneous network environment and is applicable out-of-the box to enable *personal, application and terminal mobility*. The application engine allows installing additional ODIX components from the Internet in a self-managing way and launching suitable Internet applications on-demand to process a submitted task not known in advance. The support of local, remote and distributed task processing on-demand has been shown with the integrated implementation of the application workbench, factory and federation, respectively. Besides particularly implemented ODIX applications, legacy Java and native program executables can also be run to utilize existing task applications in ODIX.

6. Conclusions

In this section, the contributions and highlights of the thesis are briefly summarized. The lessons learned are described and evaluated with respect to the goals of the work. Various suggestions about future work follow afterwards.

6.1 Summary

The overall goal of this thesis has been the design and implementation of an *Autonomic Cross-Platform Operating Environment (ACOE)* to support *On-Demand Internet Computing (ODIC)*. This has been achieved and practically demonstrated by a Java realization of the *Crossware Development Kit (XDK)* and the *On-Demand Internet Computing System (ODIX)*, respectively.

The thesis has started with the illustration of the shift from resource-centric to task-centric computing and it then presented the vision of ODIC [see Chapter 2]. The unpredictable constellation of application requirements and platform capabilities has been identified as the basic challenge to launching Internet applications on-demand. It has been shown that traditional Internet computing approaches support distinct facets of ODIC in isolated application scenarios only.

As a remedy, the introduction of an ACOE has been proposed to automate the resource provisioning by introducing a self-managing integration middleware [see Chapter 3]. It decouples the setup of the application environment needed to run the application from the administration of the computing system currently employed by the user. Existing solutions have been examined that turned out to be limited in supporting the elaborated cross-platform features and self-managing behavior.

Afterwards, a related Java realization of the XDK has been presented and the autonomic application has been illustrated for distinct components [see Chapter 4]. In particular, the self-managed deployment, composition, hosting, customization, interconnection and migration of Internet applications have been separately described and compared with related work. Altogether, it has been shown that the XDK represents an integrated and customizable Java solution to establish an ACOE.

Finally, the Java realization of ODIX for supporting on-demand local task processing, remote task processing and distributed task processing in the Internet has been presented [see Chapter 5]. The related use of various components, namely the *Internet Application Workbench*, the *Internet Application Factory* and the *Internet Application Federation*, has been described, and the suitability of ODIX to support the distinct on-demand facets of ODIC has been shown.

6.2 Lessons Learned

During the work on this dissertation, various insights have been gained that explain some interesting specifics of the autonomic approach, the system implementation, the application development and the practical usage, as described below.

Autonomic Approach. The XDK has been basically designed to operate in a self-managing way according to the autonomic computing approach of IBM. For this purpose, various autonomic managers have been introduced to monitor and control the autonomic operations. In this thesis, each manager currently uses simple analyzers and deterministic planners that evaluate the runtime parameters and separately adjust the respective operation according to given configuration files. In practice, this approach has proven to be sufficient for realizing isolated autonomic features and the correct operation could be easily observed. In an integrated application system and cross-platform operating infrastructure, however, the overall setup tends to become quite complex. With a large number of autonomic assets it is not obvious to the developer how the overall system operates, e.g. which applications actually use a code component in a multi-tasking environment and how network messages are routed via different hops in a self-organized peer federation. As a result, the use of autonomic approaches greatly facilitates system handling in a changing environment. On the other hand, the developers lose control and understanding about internal operations. In practice, this makes it difficult to design and test an integrated autonomic computing system.

System Implementation. The vision of launching Internet applications on-demand has required the implementation of an integrated application system that concurrently addresses various runtime concerns. Although there are Java libraries and application frameworks for dedicated purposes, it has been difficult to integrate them into an autonomic computing system without modifying and extending their internal implementation, e.g. to add autonomic managers, sensors and effectors. As a lesson learned, legacy code components cannot be easily reconfigured into autonomic elements. Further issues have come up with the transformation of the legacy JVM into a multi-tasking environment. The lack of deterministic thread signalling and separated memory spaces imposed the introduction of a new Java thread control and object isolation approach. The support of transparent object interconnection and execution migration has also required much development effort. After all, the core implementation of XDK and ODIX has grown up to about 350.000 lines of code and 4000 Java classes. Nevertheless, the core limitations of the JVM, such as non-preemptive thread scheduling, are still in place and cannot be changed unless the JVM itself is modified. In fact, multi-tasking execution is only reasonable for cooperative applications and non-critical use cases.

Application Development. The goal of the thesis has been the support of ODIC by implementing distinct on-demand facets, such as deployment, hosting and migration on-demand. This has been achieved for Java applications following the XDK and ODIX programming models. While the development rules are easy to follow, however, it has turned out that it is hard to convince application developers to write ODIX applications and components that can be run in the proprietary ODIX environment only. The same is valid for the transformation of existing Java applications unless particular on-demand features are needed, e.g. transparent object interconnection. As a remedy for enabling on-demand task processing with legacy applications, a small wrapper has been added that handles the incoming task processing requests and generates appropriate application calls, e.g. via a command-line interface. In this context, it has become an important feature that legacy code assemblies can be easily encapsulated in crosslet archives and decorated with configuration statements describing the deployment, composition and hosting requirements. This provides a starting set of code assemblies that can be used to deploy and run existing applications on-demand. Against this background, it has become obvious that a common specification for defining the assembly labels is needed. Otherwise, every developer defines his or her own set of assemblies, the same code component is potentially deployed multiple times and there is only little or no reuse of cached assemblies.

Practical Usage. Since ODIX is based on an integrated application system, no particular installation and configuration of third-party software components is necessary other than a legacy JVM. In particular, the automated installation via Java Web Start represents an easy and elegant way to deploy ODIX on new Internet computing systems while being online. In practice, there has been no problem to run ODIX on various computing systems, and thus to deploy and launch task applications on-demand. In contrast to the supposed permanent execution of ODIX and waiting for task processing requests in the background, however, it has been necessary to restart the JVM from time to time. Due to buggy component implementations, memory leaks and deadlocks may occur and render the whole JVM unusable. In fact, a lesson is to launch legacy Java applications that do not rely on the ODIX framework in a separate process. Besides supporting on-demand task processing, it turned out that ODIX can be used to maintain general software installations on distinct computing systems, e.g. in an enterprise network. In this case, legacy software installation packages are encapsulated in a crosslet archive and distributed to a crosslet repository. On relevant computing systems, ODIX is launched in the background and checks for new crosslets. If an update is available, the user is notified and asked for permission to retrieve and perform the software update. In this sense, ODIX may be used as a cross-platform software update service and may help to deal with heterogeneous computer installations, e.g. by evaluating centrally managed user and platform profiles.

In conclusion, the Java implementations XDK and ODIX can be used out-of-the-box to deploy and run an on-demand computing infrastructure. Their self-managing operation follows the autonomic computing approach and has proven to be suitable for supporting ODIC. The particular objective to launch Internet applications on-demand could be achieved and successfully demonstrated. Besides the development and use of specific Java-based ODIX applications, the ODIX workbench, factory and federation implementations have shown that legacy Java applications and native applications likewise can be used by ODIX to offer on-demand task processing. From this point of view, it has turned out that most appliances benefit from the self-managing deployment, composition and hosting features of ODIX that do not impose modifications of existing code on application developers. In contrast, the self-managing customization, interconnection and migration features can be utilized by Java applications that follow the ODIX programming model.

6.3 Future Work

Based on the results of this thesis, the establishment of an *Autonomic Cross-Platform Operating Environment (ACOE)* represents a feasible approach to support *On-Demand Internet Computing (ODIC)*. For advancing the present solution, there are several activities and suggestions for future work, as described below.

First of all, there are various options to improve the autonomic operation of the XDK. Its self-managing implementation is still based on simple configuration policies and controlling distinct low-level autonomic managers. By using a *rules engine*, an autonomic manager could determine the next actions by following a *rule-based policy* that allows specifying complex behaviour schemes [398]. Another promising idea is to group autonomic managers and to define high-level autonomic policies, such as *utility functions* [193, 375] and *goal-based policies* [148]. This would ease the overall configuration and free developers to define lots of low-level autonomic operations. In an advanced realization, it would allow administrators to focus on enabling on-demand business needs, e.g. the definition of *Service Level Agreements (SLA)* to favor task processing requests and optimize the overall profit [393]. The introduction of security and privacy policies could close the security gap not yet addressed in this work. In particular, the collaboration of distinct low-level autonomic elements to detect and manage security frauds on the system or network level is a challenging research task in terms of using compatible security policy definitions, establishing mutual trust and coordinating the planning of autonomic actions [57]. A related issue is to enable the spatial management and dissemination of autonomic polices in a distributed environment, e.g. by introducing a *policy deployment model* as described in [90]. In particular, the setup of autonomic operations in the ODIX application federation, e.g. running in an enterprise network, could benefit from centrally organized policies. A future implementation should consider the use of *Java Management Extensions (JMX)* for including remote autonomic management features in the XDK [1]. Concerning mobile and nomadic application scenarios, the consideration of changing system environments and the adjustment of self-managing policies according to varying user goals is another direction of future work [114, 132]. Currently, the XDK takes into account the present platform configuration to set up a suitable operating environment for launching user applications. However, this is done without adjusting the actual self-managing policy, e.g. due to different user goals when working at the office, at home or on the move. In this context, the prediction of user actions is another research direction for adapting and configuring the self-managing operation of the XDK, as proposed in the *proactive computing* approach of Intel [376].

Besides refining the overall autonomic operation of the XDK, many details of the actual feature implementations could be extended to match new research ideas, ongoing software advancements and dedicated application requirements. A major idea is to adopt the widely accepted *OSGI standard* for supporting Java application deployment and composition by using individual OSGI bundles [262]. For simple Java applications, the XDK could be extended with an OSGI bundle handler to access Java classes encapsulated within an OSGI bundle. In complex scenarios, the application execution engine could also launch an external OSGI container, e.g. *Apache Felix* [13], to run an OSGI application. A related development suggestion is to integrate and apply distinct XDK features in advanced and widely used application frameworks, such as *Eclipse Equinox* and *Spring* [95, 330]. The use of semantic technologies to describe, link and resolve resources is another idea. For example, *semantic user profiles* could be applied to deduce user preferences in nomadic application scenarios by inventing a preferences ontology and using semantic languages [325]. Another future extension of ODIX is to validate mobile code when migrating execution units. The same is valid for regular code deployment among computing nodes of various application federations that are spread across distinct organizations, e.g. by using digital signatures and a *Public Key Infrastructure (PKI)* [176] as well as the introduction of a secure deployment protocol like *S-CODEP* [147]. In this context, the concurrent hosting of multiple task applications within the same JVM may also influence other applications in terms of thread scheduling, memory allocation and resource deadlocks. Concerning task application migration, load balancing and privacy issues, further research is needed. For example, when to reuse an already running JVM and when to launch a separate JVM, e.g. by grouping multiple task applications per customer in a JVM instance and tracking the state of the JVM using the *Java Monitoring and Management Extensions* [342].

Another strong focus of future work is the evaluation of ODIX for its suitability in various task processing scenarios. A current activity is the implementation of ODIX applications for processing specific media indexing tasks in the research project *CONTENTUS* [66]. The related evaluation will investigate the use of ODIX application federations to perform uncertain processing tasks with particular respect to self-managing selection, deployment and running of suitable ODIX applications. It will also examine the options to migrate lengthy media indexing tasks and apply load-balancing strategies within the federation. A different evaluation of ODIX will be conducted in the research project *MEDIAGRID* [235]. It will focus on the effort to deploy and integrate ODIX application factories in existing computing infrastructures, e.g. Grid and Cloud installations. In this context, the ODIX approach to provide an on-demand computing infrastructure is particularly complemented with the *Platform-as-a-Service (PaaS)* idea [218]. Further investigations are needed how ODIX can utilize related Cloud computing technologies, such as *Amazon EC2* or *Google AppEngine* [9, 143], and scale on various computing nodes on-demand. A further evaluation could pick up the cloud computing idea of *Software-as-a-Service (SaaS)* [220] and examine the suitability of the ODIX application workbench to provide pervasive access on user applications and documents uploaded to the cloud. Instead of using an Internet browser and server-bound rich client techniques like AJAX [293], the entire applications are downloaded to the client computer and locally executed while still retrieving the documents from a remote site.

Finally, the XDK and ODIX features may also be used to support different application scenarios, such as *autonomic personal computing* [23]. For example, an administrator of an enterprise network could install ODIX on every desktop computer and automate *remote software installations* by uploading new software updates to a common crosslet repository. A cross-platform application installer could be used to download and update specific software installations automatically while evaluating the target platform configuration, e.g. to deploy native program packages for MS Windows and Linux installations. Moreover, a public network of crosslet repositories may be established to deploy and synchronize software components between different sites. This could also be used by software vendors to upload new applications to a pay-per-download service similar to the *AppStore* invented for *Apple's iPhone* [17] or the *Java Store* by Sun [346]. Last but not least, the use of the ODIX application engine on mobile Internet devices is another topic for future work as they become more powerful over time and considering the emerging Java-capable smart phones, e.g. running *Google's Android* [142]. A promising application scenario is to run and exchange cross-platform *widgets* without using a remote crosslet repository, e.g. by directly transmitting the code components between two side-by-side located smart phones via a Bluetooth connection.

7. References

1. Abdellatif, T., Danes, A. *Automating the Management of J2EE Servers Using a JMX-based Management System*. Transactions on Systems Science and Applications. Special Issue on Self-Organizing Communications. Vol. 2, Nr. 3. pp. 289-296.

2. *Adobe AIR*
http://www.adobe.com/products/air/

3. Agarwal, M., Bhat, V., Liu, H., Matossian, V., Putty, V., Schmidt, C., Zhang, G., Zhen, L., Parashar, M., Khargharia, B., Hariri, S. *AutoMate: Enabling Autonomic Applications on the Grid.* Proc. of the 5th Intl. Active Middleware Services Workshop (AMS). IEEE 2003. pp. 48-57.

4. *Aglets*
http://www.trl.ibm.com/aglets/

5. Aksit, M. *Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision*. Proc. of the 23rd Intl. Conference on Distributed Computing Systems Workshops. IEEE 2003. pp. 84-89.

6. Al-Bar, A. Wakeman, I. *Camel: A Mobile Applications Framework.* Proc. of the Intl. Conference on Computer Networks and Mobile Computing. IEEE 2003. pp. 214-223.

7. Al-Bar, A. Wakeman, I. A Survey of Adaptive Applications in Mobile Computing. Proc. of the 21st Intl. Conference on Distributed Computing Systems (ICDCS). IEEE 2001. pp. 246-251.

8. Aldrich, J., Dooley, J., Mandelsohn, S., Rifkin, S. *Providing Easier Access to Remote Objects in Client-Server Systems*. Proc. of the Annual Hawaii Intl. Conference on System Sciences. IEEE 1998. pp. 366-375.

9. *Amazon EC2*
http://aws.amazon.com/ec2/

10. Anderson, D. P. *BOINC: A System for Public-Resource Computing and Storage*. Proc. of the Intl. Workshop on Grid Computing. IEEE. 2004. pp. 4-10.

11. Anderson, D.P., Cobb. J., Korpels, E., Lebofsky, M., Werthimer, D. *SETI@home: An Experiment in Public-Resource Computing*. Communications of the ACM. Vol. 45, Nr. 11. ACM 2002. pp. 56-61.

12. *AntHill*
http://www.cs.unibo.it/projects/anthill/

13. *Apache Felix*
http://felix.apache.org

14. *Apache Maven*
http://maven.apache.org

15. *Apache Server Framework Avalon*
http://jakarta.apache.org/avalon/framework/index.html

16. *Apache Tomcat*
http://jakarta.apache.org.

17. *Apple AppStore*
http://www.apple.com/iphone/appstore/

18. Astley, M., Sturman, D., Agha, G. *Customizable Middleware for Modular Distributed Software*. Communications of the ACM. Vol. 44, Nr. 5. ACM 2001. pp. 99-107.

19. *Autonomous Remote Cooperating Agents*.
http://osweb.iit.unict.it/ARCA

20. Avvenuti, M., Vecchio, A. *Embedding Remote Object Mobility in Java RMI.* Proc. of the 8th Intl. Workshop on Future Trends of Distributed Computing Systems (FTDCS 2001). IEEE 2001. pp. 98-104.

21. Balasubramanian1, J., Natarajan, B., Schmidt, D. C., Gokhale, A., Parsons, J., Deng, G. *Middleware Support for Dynamic Component Updating*. Proc. of the 7th Intl. Symposium on Distributed Objects and Applications (DOA). LNCS 3761. Springer 2005. pp. 978-996.

22. Balfanz, D., Gong, L. *Experience with Secure Multi-Processing in Java*. Proc. of the Intl. Conference on Distributed Computing Systems (ICDCS). IEEE 1998. pp. 398-405.

23. Bantz, D. F., Bisdikian, C., Challener, D., Karidis, J. P., Mastrianni, S., Mohindra, A., Shea, D. G., Vanover, M. *Autonomic Personal Computing*. IBM Systems Journal. Vol. 42, Nr. 1. IBM 2003. pp. 165-176.

24. Barak, A., La'adan O. *The MOSIX Multicomputer Operating System for High Performance Cluster Computing*. Journal of Future Generation Computer Systems. Vol. 13, No. 4-5. Elsevier 1998. pp. 361-372.

25. Barett, R. Maglio, P. P. *Intermediaries: An Approach to Manipulating Information Streams*. IBM System Journal. Vol. 38, Nr. 4. IBM 1999. pp. 629-641.

26. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. *Xen and the Art of Virtualization*. Proc. of the 19th ACM Symposium on Operating Systems Principles. ACM 2003. pp. 164-177.

27. Barr, M., Eisenbach, S. *Safe Upgrading without Restarting*. Proc. of the Intl. Conference on Software Maintenance. IEEE 2003. pp. 129-137.

28. Batista, T., Rodriguez, N. *Dynamic Reconfiguration of Component-Based Applications*. Proc. of the Intl. Symposium on Software Engineering for Parallel and Distributed Systems. IEEE 2000. pp. 32-39.

29. Bay, T. G., Eugster, P., Oriol, M. *Generic Component Lookup*. Proc. of the Intl. Conference on Component-Based Software Engineering. LNCS 4063. Springer 2006. pp. 182-197.

30. Bellavista, P., Corradi, A., Montanari, R., Stefanelli, C. *Dynamic Binding in Mobile Applications*. IEEE Internet Computing. Vol. 7, Nr. 2. IEEE 2003. pp. 34-42.

31. Benslimane, D., Dustdar, S., Sheth, A. *Services Mashups - The New Generation of Web Applications*. IEEE Internet Computing. Vol. 12, Nr. 5. IEEE 2008. pp. 13-15.

32. *Berkeley Open Infrastructure for Network Computing (BOINC)* http://boinc.berkeley.edu/

33. Bettini, L., Nicola, R. *Translating Strong Mobility into Weak Mobility*. Proc. of the 5th Intl. Conference on Mobile Agents. LNCS 2240. Springer 2001. pp. 182-197.

34. Bettini, L. *A Java Package for Transparent Code Mobility*. Proc. of the Intl. Workshop on Scientific Engineering of Distributed Java Applications. LNCS 3409. Springer 2005. pp. 112-122.

35. Bialek, R., Jul, E., Schneider, J.-G., Jin, Y. *Partitioning of Java Applications to Support Dynamic Updates*. Proc. of the Asia-Pacific Software Engineering Conference (APSEC'04). IEEE 2004. pp. 616-623.

36. Binder, W, Hulaas, J., Villazón, A., Vidal, R. *Portable Resource Control in Java: The JSEAL2 Approach.* Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001). ACM 2001. pp. 139-155.

37. Blair, G. S., Blair, L. Issarny, V., Tuma, P., Zarras, A. *The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms*. Middleware 2000. LNCS 1795. Springer 2000. pp. 164-184.

38. Blair, G. S., Costa, F., Coulson, G., Delpiano, F., Duran, H., Dumant, B., Horn, F., Parlavantzas, N., Stefani, J.-B. *The Design of a Resource-Aware Reflective Middleware Architecture*. Meta-Level Architecture and Reflection 1999. LNCS 1616. Springer 1999. pp. 115-134.

39. Blair, G. S., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N. *Reflection, Self-Awareness and Self-Healing in OpenORB*. Proc. of the First Workshop on Self-Healing Systems. ACM 2002. pp. 9-14.

40. Blair, G. S., Coulson, G., Robin, P., Papathomas, M. *An Architecture for Next Generation Middleware*. Proc. of Intl. Conferenece on Distributed Systems Platforms and Open Distributed Processing (Middleware 98). Springer 1998. pp. 191-206.

41. Blau, J. *Microsoft: Community Computing is on the Way*. IDG News Service. 22.11.2005. http://www.networkworld.com/news/2005/112205-community-computing.html

42. Blohm, H. *Hierarchical Arrangement of Modified Class Loaders and Token Driven Class Loaders and Methods of Use*. United States Patent 7.536.412.B2. 2009.

43. Bouchenak, S., Hagimont, D. *Pickling Threads State in the Java System*. Proc. of the Intl. Conference on Technology of Object-Oriented Languages (TOOLS). IEEE 2000. pp. 22-32.

44. Braun, P. *The Migration Process of Mobile Agents - Implementation, Classification, and Optimization*. PhD Thesis. Friedrich-Schiller-Universität Jena 2003.

45. Braun, P. Rossak, W. *Mobile Agents - Basic Concepts, Mobility Models, and the Tracy Toolkit*. Elsevier 2005.

46. Brazier, F.M.T., Overeinder, B.J., Steen, M., van Wijngaards, N.J.E. *Supporting Internet-Scale Multi-Agent Systems*. Data & Knowledge Engineering. Vol. 41, Nr. 2-3. Elsevier 2002. pp. 229-245.

47. Bouchenak, S., Hagimont, D., Krakowiak, S., Palma, N., Boyer, F. *Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence*. Software - Practice & Experience. Vol. 34, Nr. 4. Wiley & Sons 2004.

48. Brazier, F. M. T., Kephart, J. O., Van Dyke Parunak, H., Huhns, M. N. *Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda*. IEEE Internet Computing. Vol. 13, Nr. 3. IEEE 2009. pp. 82-87.

49. Buckley, A. *JSR 294: Improved Modularity Support in the Java Programming Language*. Sun Microsystems. http://jcp.org/en/jsr/detail?id=294

50. Bursell, M., Hayton, R., Donaldson, D., Herbert, A. *A Mobile Object Workbench*. Proc. of the Intl. Workshop on Mobile Agents. LNCS 1477. Springer 1998. pp. 136-147.

51. Carlegren, F., Diaz, A. L., McCrimmon, T. *Operating Environment Essentials for an On Demand Breakthrough*. IBM DeveloperWorks. IBM 2004. http://www-106.ibm.com/developerworks/library/i-odoe2/

52. Carzaniga, A, Picco, G.P., Vigna, G. *Designing Distributed Applications with Mobile Code Paradigms*. Proc. of the 19th Intl. Conference on Software Engineering. (ICSE'97). ACM Press 1997. pp. 22-32.

53. Caughey, S. J., Hagimont, D., Ingham, D. B. *Deploying Distributed Objects on the Internet.* Advances in Distributed Systems. LNCS 1752. Springer 2000. pp. 213-237.

54. Chakravarti, A., Wang, X., Hallstrom, J. O., Baumgartner, G. *Implementation of Strong Mobility for Multi-Threaded Agents in Java*. Intl. Conference on Parallel Processing. IEEE 2003. pp. 321-330.

55. Chen, X., Simons, M. *A Component Framework for Dynamic Reconfiguration of Distributed Systems*. Proc. of the Intl. Conference on Component Deployment (CD 2002). LNCS 2370. Springer 2002. pp. 82-96.

56. Chess, D. M. *Security Issues in Mobile Code Systems*. Mobile Agents and Security. LNCS 1419. Springer 1998. pp. 1-14.

57. Chess, D. M., Palmer, C. C., White, S. R. *Security in an Autonomic Computing Environment*. IBM Systems Journal. Vol. 42, Nr. 1. IBM 2003. pp. 5-18.

58. Chester, T. M. Cross-*Platform Integration with XML and SOAP*. IEEE IT Professional. Vol. 3, Nr. 5. IEEE 2001. pp. 26-34.

59. Chien, A., Calder, B., Elbert, S., Bhatia, K. *Entropia: Architecture and Performance of an Enterprise Desktop Grid System*. Journal of Parallel and Distributed Computing. Vol. 63, Nr. 5. Elsevier 2003. pp. 597-610.

60. Clark, D. *Face-to-Face with Peer-to-Peer Networking*. IEEE Computer. Vol. 34, Nr. 1. IEEE 2001. pp. 18-21.

61. Clark, D. *Network Nirvana and the Intelligent Device*. IEEE Concurrency. Vol. 7, Nr. 2. IEEE 1999. pp. 16-19.

62. Clarke, M., Blair, G. S., Coulson, G., Parlavantzas, N. *An Efficient Component Model for the Construction of Adaptive Middleware*. Proc. of the Intl. Conference on Middleware 2001. LNCS 2218. Springer 2001. pp. 160-178.

63. Cohen, G. A., Chase, J. S., Kaminsky, D. L. *Automatic Program Transformation with JOIE.* Proc. of the USENIX Annual Technical Symposium. USENIX 1998. pp. 167-178.

64. *Condor - High Throughput Computing*
http://www.cs.wisc.edu/condor/

65. di Constanzo, A., de Assuncao, M., D., Buyya, R. *Harnessing Cloud Technologies for a Virtualized Distributed Computing Infrastructure*. IEEE Internet Computing. Vol. 13, Nr. 5. pp. 24-33.

66. *CONTENTUS*
http://www.iais.fraunhofer.de/contentus.html

67. Coombes, R., Siddiqi, J. *A Framework for IT as a Utility*. Proc. of the Intl. Conference on Information Technology: New Generation. IEEE 2008. pp. 218-223.

68. Cornell, G., Horstmann, C. S. *Core Java*. SunSoft Press 1996.

69. Corwin, J., Bacon, D.F., Grove, D., Murthy, C. *MJ: A Rational Module System for Java and its Applications*. Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). ACM 2003. pp. 241-254.

70. Costa, F., Blair, G. S., Coulson, G. *Experiments with Reflective Middleware*. Proc. of the ECOOP 1998. LNCS 1543. Springer 1998. pp. 390-391.

71. Coulson, G., Blair, G. S., Clarke, M., Parlavantzas, N. *The Design of a Configurable and Reconfigurable Middleware Platform*. Distributed Computing. Vol. 15, Nr. 2. Springer 2002. pp. 109-126.

72. Cunsolo, V. D., Distefano, S., Puliafito, A., Scarpa, M. *Cloud@Home - Bridging the Gap between Volunteer and Cloud Computing*. Proc. of the Intl. Conference on Intelligent Computing. LNCS5754. Springer 2009. pp. 423-432.

73. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S. *Unraveling the Web Services Web. An Introduction to SOAP, WSDL, and UDDI.* IEEE Internet Computing. Vol. 6, Nr. 2. IEEE 2002. pp. 86-93.

74. Czajkowski, G. *Application Isolation in the Java Virtual Machine.* Proc. the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM 2000. pp. 354-366.

75. Czajkowski, G., Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution.* Proc. the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM 2001. pp. 125-138.

76. Czajkowski, G., Daynes, L., Nystrom, N. *Code Sharing among Virtual Machines*. Proc- of the European Conference on Object-Oriented Programming (ECOOP). ACM 2002. pp. 155-177.

77. Czajkowski, G., Daynes, L., Titzer, B. *A Multi-User Virtual Machine.* Proc. of the USENIX 2003 Annual Technical Conference. USENIX 2003. pp. 85-98.

78. Czajkowski, G., Hahn, S., Skinner, G., Soper, P., Bryce, C. *A Resource Management Interface for the Java Platform*. TR-2003-24. Sun Microsystems 2003.

79. Damianou, N., Dulay, N., Lupu, E., Sloman, M. *The Ponder Policy Specification Language*. Proc. of the 2nd Intl. Workshop on Policies for Distributed Systems and Networks (Policy'01). LNCS 1995. Springer 2001. pp. 18-38.

80. David, P.-C., Ledoux, T. *An Infrastructure for Adaptable Middleware.* Prof. of the 4th. Intl. Symposium on Distributed Objects and Applications (DOA). LNCS 2519. Irvine, USA. Springer 2002. pp. 773-790.

81. Daynes, L., Czajkowski, G. *Sharing the Runtime Representation of Classes across Class Loaders*. Proc. of the European Conference on Object-Oriented Programming (ECOOP). LNCS 3586. Springer 2005. pp. 97-120.

82. Dearle, A. *Toward Ubiquitous Environments for Mobile Users*. IEEE Internet Computing. Vol. 2, Nr. 1. IEEE 1998. pp. 22-32.

83. Dearle, A., Kirby, G. N. C. , McCarthy, A., Diaz y Carballo, J. C. *A Flexible and Secure Deployment Framework for Distributed Applications*. Proc. of the Intl. Conference on Component Deployment (CD 2004). LNCS 3083. Springer 2004. pp. 219-233.

84. *Deploy Directory.*
http://www.quest.com/deploydirector/.

85. Diaz y Carballo J.C., Dearle A., Connor R.C.H. *Thin Servers - An Architecture to Support Arbitrary Placement of Computation in the Internet*. Proc. of the Intl. Conference on Enterprise Information Systems (ICEIS 2002). ICEIS 2002. pp. 1080-1085.

86. Dikaiakos, M. D., Pallis, G., Katsaros, D., Mehra, P., Vakali. A. *Cloud Computing - Distributed Internet Computing for IT and Scientific Research*. IEEE Internet Computing. Vol. 13, Nr. 5. IEEE 2009. pp. 10-13.

87. Dillenberger, W., Bordwekar, R., Clark, C., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M., Samuel, F., St. John, R. *Building a Java virtual machine for server applications: The JVM on OS/390*. IBM Systems Journal. Vol. 39, No 1. 2000. pp. 194-210.

88. *DistrIT*
http://distrit.sourceforge.net/

89. Dittmar, T. *Ad-Hoc Migration aktiver Java Komponenten in einem verteilten Anwendungs-system*. Diploma Thesis. University of Marburg 2005.

90. Dulay, N., Lupu, E., Sloman, M., Damianou, N. *A Policy Deployment Model for the Ponder Language*. Proc. of the Intl. Symposium on Integrated Network Management. IEEE 2001. pp. 14-18.

91. Durnant, B. Tran, F., Horn, F., Stefani, J. B. *Jonathan: An Open Distributed Processing Environment in Java*. Proc. of the Intl. Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98). Springer 1998. pp. 175-190.

92. Eberhard, J., Tripathi, A. *Efficient Object Caching for Distributed Java RMI Applications*. Proc. of the Intl. Conference on Middleware. LNCS 2218. Springer 2001. pp. 15-35.

93. *Echidna*
http://ostatic.com/echidna

94. Eckel, B. *Thinking in Java*. Prentice Hall 2000.

95. *Eclipse Equinox*
http://www.eclipse.org/equinox/

96. *Eclipse Rich Client Platform (RAP)*
http://www.eclipse.org/rcp/

97. *Eclipse Rich AJAX Platform (RAP)*
http://www.eclipse.org/rap/

98. Edwards, W. K., Grinter, R. E. *At Home With Ubiquitous Computing - Seven Challenges*. Proc. of the Intl. Conference on Ubiquitous Computing. LNCS 2201. Springer 2001. pp. 256-272.

99. Einstein@home
http://einstein.phys.uwm.edu/

100. Eisenbach, S., Kayhan, D., Sadler, C. *Keeping Control of Reusable Components*. Proc. of the Intl. Working Conference on Component Deployment (CD 2004). LNCS 3083. Springer 2004. pp. 144-158.

101. Ellahi, T. N., Hudzia, B., McDermott, L., Kechadi, T. *Transparent Migration of Multi-Threaded Applications on a Java Based Grid*. Prof. of the Intl. Conference on Web Technologies, Applications, and Services (WTAS 2006). IASTED 2006.

102. Erwin, D., Snelling, D. *UNICORE: A Grid Computing Environment*. Proc. of the Intl. Conference Euro-Par. LNCS 2150. Springer 2001. pp. 825-834.

103. *Eucalyptus*
http://open.eucalyptus.com/

104. Eugster, P. T., Baehni, S. *Abstracting Remote Object Interaction in a Peer-to-Peer Environment.* Proc. of the Intl. Symposium on Computing in Object-oriented Parallel Environments (ISCOPE). ACM 2002. pp. 46-55.

105. *Exymen - Extend Your Media Editor Now*
http://www.exymen.org

106. *eyeOS - Cloud Computing Platform*
http://www.eyeos.org

107. *FairThreads - Framework for Concurrent and Parallel Programming*
http://www-sop.inria.fr/mimosa/rp/FairThreads/index.html

108. Farley, J. *Microsoft .NET vs. J2EE: How Do They Stack Up*. O'Reilly 2001.

109. Fedak, G., Germain, C., Neri, V., Cappello, F. *XtremWeb: A Generic Global Computing Platform*. Proc. of the Intl. Symposium on Cluster Computing and the Grid. IEEE 2001. pp. 582-587.

110. Fellenstein, C. *On Demand Computing - Technologies and Strategies*. IBM Press. 2005.

111. Figueiredo, R., Dinda, P., Fortes, J. *A Case for Grid Computing on Virtual Machines*. Proc. of the Intl. Conference on Distributed Computing Systems (ICDCS). IEEE 2003. pp. 550-559.

112. Flammia, G. *Peer-to-Peer Is Not For Everyone*. IEEE Intelligent Systems. Vol. 16, Nr. 3. IEEE 2001. pp. 78-79.

113. Fleury, M., Reverbel, F. *The JBoss Extensible Server.* Proc. of the ACM Intl. Middleware Conference. LNCS 2672. Springer 2003. pp 344-373.

114. Floch, J., Stav, E., Hallsteinsen, S. *Interfering Effects of Adaptation: Implications on Self-Adapting Systems Architecture*. Proc. of the Intl. Conference on Distributed Applications and Interoperable Systems (DOA). LNCS 4025. Springer 2006. pp. 64-69.

115. *Folding@HOME*
http://folding.stanford.edu/

116. Foster, I. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. Proc. of the 1st Intl. Symposium on Cluster Computing and the Grid. IEEE 2001. pp. 6-7.

117. Foster, I., Kesselman, C. *Globus: A Metacomputing Infrastructure Toolkit*. Intl. Journal of Supercomputer Applications. Vol. 11, Nr. 2. MIT Press 1997. pp. 115-128.

118. Foster, I., Kesselman, C., Nick, J. M., Tuecke, S. *Grid Services for Distributed System Integration.* IEEE Computer. Vol. 35, Nr. 6. IEEE 2002. pp. 37-46.

119. Foster, I., Zhao, Y., Raicu, I., Lu, S. *Cloud Computing and Grid Computing 360-Degree Compared*. Proc. of the Intl. Workshop on Grid Computing Environments IEEE 2008. pp. 1-10.

120. *Foundation for Intelligent Physical Agents (FIPA)*
http://www.fipa.org/

121. Fox, G. *Peer-to-Peer Networks*. IEEE Computing in Science & Engineering. Vol. 3, Nr. 3. IEEE 2001. pp. 75-77.

122. Fox, G., Gannon, D. *Computational Grids*. IEEE Computing in Science & Engineering. Vol. 3, Nr. 4. IEEE 2001. pp. 74-77.

123. Franz, M. *Dynamic Linking of Software Components*. IEEE Computer. Vol. 30, Nr. 3. IEEE 1997. pp. 74-81.

124. Fraser, K. A., Hand, S. M., Harris, T. L., Leslie, I. M., Pratt, I. A. *The XenoServer Computing Infrastructure*. Technical Report UCAM-CL-TR-552. University of Cambridge, Computer Laboratory. January 2003.

125. Friese, T., Smith, M., Freisleben, B. *Hot Service Deployment in an Ad Hoc Grid Environment*. Proc. of the Intl. Conference on Service-Oriented Computing. ACM 2004. pp. 75-83.

126. Fugetta, A. Picco, G.P., Vigna, G. *Understanding Code Mobility.* IEEE Transactions on Software Engineering. Vol. 24, Nr. 5. IEEE 1998. pp. 342-361.

127. Fünfrocken, S. *Transparent Migration of Java-based Mobile Agents: Capturing and Reestablishing the State of Java Programs.* Proc. of the 2nd Intl. Workshop on Mobile Agents. LNCS 1477. Springer 1998. pp. 26-37.

128. Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., Woodall, T. S. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation.* In 11th European PVM/MPI. LNCS 3241. Springer, 2004. pp. 97-104.

129. Gamma, E., Helm, R, Johnson, R., Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995.

130. Ganek, A. G., Corbi, T. A. *The Dawning of the Autonomic Computing Era*. IBM Systems Journal. Vol. 42, Nr. 1. IBM 2003. pp. 5-18.

131. Gavalda, C. P., Lopez, P. G., Andreu, R., M. *Deploying Wide-Area Applications Is a Snap*. IEEE Internet Computing. Vol. 11, Nr. 2. IEEE 2007. pp. 72-79.

132. Geihs, K. *Selbst-Adaptive Software.* Informatik Spektrum. Springer 2007. pp. 133-145.

133. Gil, J.-M., Choi, S.-J. *A Peer to Peer Grid Computing System Based on Mobile Agents*. Proc. of the Intl. Conference on Agent and P2P Computing. LNAI 4461. Springer 2008. pp. 175-186.

134. *gLite - Lightweight Middleware for Grid Computing*
http://glite.web.cern.ch/glite/

135. *The Globus Project*
http://www.globus.org

136. Godfrey, B., Lakshminarayanan, K., Surana, S. Karp, R., Stoica, I. *Load Balancing in Dynamic Structured P2P Systems*. P2P Computing Systems. Vol. 63, Nr. 3. Elsevier 2006. pp. 217-240.

137. Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft, P. *SmartFrog: Configuration and Automatic Ignition of Distributed Applications.* HP Labs, Bristol, UK. http://www.hpl.hp.com/research/smartfrog/

138. Golm, M., Felser, M., Wawersich, C. *The JX Operating System*. Proc. of the USENIX Annual Technical Conference. 2002. pp. 45-58.

139. Gong, L. *Secure Java Class Loading*. IEEE Internet Computing. Vol. 2, Nr. 6. IEEE 1998. pp. 56-61.

140. Gong, L. *JXTA: A Network Programming Environment*. IEEE Internet Computing. Vol. 5, Nr. 3. IEEE 2001. pp. 88-95.

141. Goodwill, J. *Apache Jakarta Tomcat.* APress. 2001.

142. *Google Android*
http://www.android.com/

143. *Google AppEngine*
http://code.google.com/appengine/

144. *Google Web Toolkit*
http://code.google.com/webtoolkit/

145. Graupner, S., Kotov, V., Andrzejak, A., Trinks, H. *Service-Centric Globally Distributed Computing*. IEEE Internet Computing. Vol. 7, Nr. 4. IEEE 2003. pp. 36-43.

146. *Great Internet Prime Search (GIMPS)*
http://www.mersenne.org/

147. Grechanik, M., Perry, D. E. *Secure Deployment of Components*. Proc. of the Intl. Conference on Component Deployment (CD 2004). LNCS 3083. Springer 2004. pp. 175-189.

148. Greenwood, D., Rimassa, G. *Autonomic Goal-Oriented Business Process Management*. Proc. of the Intl. Conference on Autonomic and Autonomous Systems. IEEE 2007. pp. 43-48.

149. *GridGain*
http://www.gridgain.com/

150. Grossmann, R. L. *The Case for Cloud Computing*. IEEE IT Professional. Vol. 11, Nr. 2. IEEE 2009. pp. 23-27.

151. Grimshaw, A., Ferrari, A., Knabe, F., Humphrey, M. *Wide-Area Computing: Resource Sharing on a Large Scale*. IEEE Computer. Vol. 32, Nr. 5. IEEE 1999. pp. 29-37.

152. Grimshaw A. S., Wulf W. A. *The Legion Vision of a Worldwide Virtual Computer*. Communications of the ACM. Vol. 40, Nr. 1. ACM 1997. pp. 39-45.

153. Griss, M. L., Pour, G. *Accelerating Development with Agent Components*. IEEE Computer. Vol. 34, Nr. 5. IEEE 2001. pp. 37-43.

154. Gruber, O., Hargrave, B. J., McAffer, J., Rapicault, P., Watson, T. *The Eclipse 3.0 Platform: Adopting OSGi Technology*. IBM Systems Journal. Vol. 44, Nr. 2. IBM 2005. pp. 289-299.

155. Grundy, J. *Storage and Retrieval of Software Components Using Aspects*. Proc. of 23rd Australasian Computer Science Conference. IEEE 2000. pp. 95-103.

156. Gupta, R., Talwar, S., Agrawal, D. P. *Jini Home Networking: A Step Toward Pervasive Computing*. IEEE Computer. Vol. 35, Nr. 8. IEEE 2002. pp. 34-40.

157. Guy-Ari, G. *Empower RMI with TRMI*. JavaWorld. Nr. 9. IDG 2002. http://www.javaworld.com/javaworld/jw-08-2002/jw-0809-trmi_p.html

158. Hall, M. *Core Servlets and JavaServer Pages (JSP)*. Prentice Hall/Sun Microsystems Press 2000.

159. Hall, R. S. *A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks*. Proc. of the Intl. Workshop on Component Deployment (CD 2004). LNCS 3083. Springer 2004. pp. 81-96.

160. Hall, R. S., Heimbigner, D, Wolf, A. L. *A Cooperative Approach to Support Software Deployment Using the Software Dock.* Proc. of the 21st Intl. Conference on Software Engineering (ICSE 1999). Los Angeles, USA. ACM 1999. pp. 174-183.

161. Hallenborg, K., Kristensen, B. B. *Jini Supporting Ubiquitous and Pervasive Computing*. Proc. of the 5th Intl. Symposium on Distributed Objects and Applications (DOA). LNCS 2888. Springer 2003. pp. 1110-1132.

162. Hammel, M. *Dynamische Lokalisierung und Bindung von migrierenden Web Services*. Diploma Thesis. University of Siegen 2002.

163. Hanssen, O., Eliassen, F. *A Framework for Policy Bindings*. Proc. of the Intl. Conference on Distributed Objects and Applications (DOA 1999). IEEE 1999. pp. 2-11.

164. Har'El, Z., Rosberg, Z. *Java Class Broker - A Seamless Bridge from Local to Distributed Programming*. Journal of Parallel and Distributed Computing. Vol. 60, Nr. 10. Elsevier Science 2000. pp. 1223-1237.

165. Hawblitzel, C., Chang, C-C., Czajkowski, G., Hu, D., von Eicken, T. *Implementing Multiple Protection Domains in Java*. Proc. of the USENIX Annual Conference. 1998. pp. 22.

166. Hayton, R., Herbert, A. *FlexiNet: A Flexible, Component-Oriented Middleware System.* Advances in Distributed Systems. LNCS 1752. Springer 2000. pp. 497-508.

167. Helal, S. *Pervasive Java*. IEEE Pervasive Computing. Vol. 1, Nr. 1. IEEE 2002. pp. 82-85.

168. Holder, O., Ben-Shaul, I., Gazit, H. *Dynamic Layout of Distributed Applications in FarGo*. Proc. of the 21st Intl. Conference on Software Engineering. ACM 1999. pp. 163-173.

169. Huebscher, M.C., McCann, J.A. *A Survey of Autonomic Computing - Degrees, Models, and Applications*. ACM Computing Surveys. Vol 40, Nr. 3. ACM 2008. pp. 1-31.

170. Huhns, M. N., Singh, M. P. *Service-Oriented Computing: Key Concepts and Principles*. IEEE Internet Computing, Vol. 9, Nr. 1. IEEE 2005. pp. 75-81.

171. Hunter, J., Crawford, W., Ferguson, P. *Java Servlet Programming*. O'Reilly & Associates 1998.

172. *IBM World Community Grid*
http://www.worldcommunitygrid.org/

173. Illmann, T., Kruger, T., Kargl, F., Weber, M. *Transparent Migration of Mobile Agents using the Java Platform Debugger Architecture*. Proc. of the Intl. Conference on Mobile Agents. LNCS 2240. Springer 2001. pp. 198-212.

174. *IoMega. v.Clone - Take your PC Virtually Anywhere*
http://protection-suite.iomega-web.com/vclone

175. Ishida, T. *Towards Computation over Communities*. Community Computing and Support Systems. LNCS 1519. Springer 1998. pp. 1-10.

176. Ismail, L. *Authentication Mechanisms for Mobile Agents*. Proc. of the Intl. Conference on Availability, Reliability and Security. IEEE 2007. pp. 246-253.

177. Izatt, M., Chan, P. *Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications.* Java Grand Conference. ACM 1999. pp. 15-24.

178. *JADE*
http://sharon.cselt.it/projects/jade/

179. *JBOSS Application Server*
http://www.jboss.org

180. *JNode - Java New Operating System Design Effort*
http://www.jnode.org

181. *JOS - Java Operating System*
http://jos.sourceforge.net

182. Jsh - The Open Source Java Application Launcher
http://gerard.collin3.free.fr/index.html

183. JPPF - Java Parallel Processing Framework
http://www.jppf.org

184. *JX - The Fast and Flexible Java OS*
http://www.jxos.org/

185. Jeon, H., Petrie, C., Cutkosky, M. R. *JATLite: A Java Agent Infrastructure with Message Routing*. IEEE Internet Computing. Vol. 4, Nr. 2. IEEE 2000. pp. 87-96.

186. Jiang, X., Xu, D. *SODA: A Service-On-Demand Architecture for Application Service Hosting Utility Platforms*. Proc. of the Conference on High Performance Distributed Computing (HPDC). Seattle, USA. IEEE 2003. pp. 174-183.

187. *Jonas*
http://jonas.ow2.org

188. Kebbal, D., Bernard, G. *Component Search Service and Deployment of Distributed Applications*. Proc. of the 3rd Intl. Symposium on Distributed Objects and Applications (DOA). IEEE 2001. pp. 125-134.

189. Keith, W. *Core Jini*. Prentice Hall 1999.

190. Keller, A., Badonnel, R. *Automating the Provisioning of Application Services with the BPEL4WS Workflow Language*. Proc. of the Intl. Conference on Utility Computing. LNCS 3278. Springer 2004. pp. 15-27.

191. Kephart, J. O., Chess, D. M. *The Vision of Autonomic Computing*. IEEE Computer. Vol. 36, Nr. 1. IEEE 2003. pp. 41-50.

192. Kephart, J. O. *Research Challenges of Autonomic Computing*. Proc. of the Intl. Conference on Software Engineering (ICSE). ACM 2005. pp. 15-22.

193. Kephart, J. O., Das, R. *Achieving Self-Management via Utility Functions*. Internet Computing. Vol. 11, Nr. 1. IEEE 2007. pp. 40-47.

194. Kern, S., Braun, P., Fensch, C., Rossak, W. *Class Splitting as a Method to Reduce Migration Overhead of Mobile Agents*. Proc. of the Intl. Symposium on Distributed Objects and Applications (DOA). LNCS 3291. Springer 2004. pp. 1358-1375.

195. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes C., Loingtier, J.-M., and Irwin, J. *Aspect-Oriented Programming*. Proc. of the European Conference on Object-Oriented Programming (ECOOP). LNCS 1241. Springer 1997. pp. 220-242.

196. Kindberg, T., Barton J. *A Web-Based Nomadic Computing System*. Pervasive Computing. Vol. 34, Nr. 4. Elsevier 2001. pp. 443-456.

197. Kindberg, T., Fox, A. *System Software for Ubiquitous Computing*. IEEE Pervasive Computing. Vol. 1, Nr. 1. IEEE 2002. pp. 70-81.

198. Kleinrock, L. Nomadic Computing - An Opportunity. ACM SIGCOMM Computer Communication Review. Vol. 25, Nr. 1. ACM 1995. pp. 36-40.

199. Kleinrock, L. *Nomadic Computing and Smart Spaces*. IEEE Internet Computing. Vol. 4, Nr. 1. IEEE 2000. pp. 52-53.

200. Kniesel, G., Costanza, P., Austermann, M. *JMangler - A Framework for Load-Time Transformation of Java Class Files*. Proc. of the Workshop on Source Code Analysis and Manipulation. IEEE 2001. pp. 100-110.

201. Kon, F., Campbell, R. H. *Dependence Management in Component-Based Distributed Systems*. IEEE Concurrency. Vol. 8, Nr. 1. IEEE 2000. pp. 26-36.

202. Kortuem, G, Fickas, S., Segall, Z. *On-Demand Delivery of Software in Mobile Environments*. Proc. of the Nomadic Computing Workshop. 1997.

203. Koster, R., Black, A. P., Huang, J., Walpole, J., Pu, C. *Thread Transparency in Information Flow Middleware*. Middleware 2001. LNCS 2218. Springer 2001. pp. 121-140.

204. Kotsovinos, E. *Global Public Computing*. Technical Report. University of Cambridge 2005.

205. Koutsonikola, V., Vakali, A. *LDAP: Framework, Practices, and Trends*. IEEE Internet Computing. Vol. 8, Nr. 5. IEEE 2004. pp. 66-72.

206. Kozuch, M., Satyanarayanan, M., Bressoud, T., Helfrich, C., Sinnamohideen, S. *Seamless Mobile Computing on Fixed Infrastructure*. IEEE Computer. Vol. 32, Nr. 7. IEEE 2004. pp. 65-72.

207. Kumara, M. H. W., He, P., Sun, X. *An Agent-Based Approach for Universal Personal Computing*. Proc. of the Intl. Conference on Circuits and Systems. IEEE 2000. pp. 18-21.

208. Kuperberg, M., Krogmann, K., Reussner, R. *Performance Prediction for Black-Box Components Using Reengineered Parametric Behaviour Models*. Proc. of the Intl. Conference on Component Based Software Engineering. LNCS 5282. Springer 2008. pp. 48-63.

209. Kurzyniec, D., Sunderam, V. *Flexible Class Loader Framework: Sharing Java Resources in Harness System.* Proc. of the Intl. Conference on Computer Science. LNCS 2073. Springer 2001. pp. 375-384.

210. Kurzyniec, D., Wrzosek, T., Sunderam, V., Sominski,A. *RMIX: A Multiprotocol RMI Framework for Java.* Proc. of the Parallel and Distributed Processing Symposium (IPDPS'03). IEEE 2003. pp. 140.

211. Lai, A., Nieh, J. *Limits of Wide-Area Thin-Client Computing*. Proc. of the Intl. Conference on Measurements and Modeling of Computer Systems. ACM 2002. pp. 228-239.

212. Lange, D. B. Oshima, M. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley 1998.

213. von Laszewski, G., Blau, E., Bletzinger, M., Gawor, J., Lane, P., Martin, S., Russel M. *Software, Component, and Service Deployment in Computational Grids*. LNCS 2370. Springer 2002. pp. 244-256.

214. Lau, K.-K., Wang, Z. *A Taxonomy of Software Component Models*. Proc. of the EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE 2005. pp. 88-95.

215. Lawton, G. *Developing Software Online With Platform-as-a-Service Technology*. IEEE Computer. Vol. 41, Nr. 6. IEEE 2008. pp. 13-15.

216. Lawton, G. *New Ways to Build Rich Internet Applications*. IEEE Computer. Vol. 41, Nr. 8. IEEE 2008. pp. 10-12.

217. Lea, D. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley 1999.

218. Leavitt, N. *Is Cloud Computing Really Ready for Prime Time*. IEEE Computer. Vol. 42, Nr. 1. IEEE 2009. pp. 15-20.

219. Lee, C., Lim, S. H., Lim, S. S., Park, K. H. *Autonomous Management of Clustered Server Systems Using JINI.* Proc. of the Intl. Conference on Utility Computing. LNCS 3278. Springer 2004. pp. 124-134.

220. Lenk, A., Klems, M., Nimis, J., Tai, S., Sandholm, T. *What's inside the Cloud? An Architectural Map of the Cloud Landscape*. Proc. of the Intl. Workshop on Software Engineering Challenges of Cloud Computing. IEEE 2009. pp. 23-31.

221. Li, Y., Leung, V. C. M. *Supporting Personal Mobility for Nomadic Computing over the Internet*. ACM SIGMOBILE Mobile Computing and Communications Review. Vol. 1, Nr. 1. ACM 1997. pp. 22-31.

222. Li, Y., Leung, V. C. M. *A Framework for Universal Personal Computing*. Proc. of the Intl. Conference on Universal Personal Communication. IEEE 1996. pp. 769-773.

223. Liang, S., Bracha, G. *Dynamic Class Loading In The Java Virtual Machine.* Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM 1998. pp. 36-44.

224. Linaje, M., Preciado, J. C., Sanchez-Figueroa, F. *Engineering Rich Internet Application User Interfaces over Legacy Web Models*. IEEE Internet Computing. Vol. 11, Nr. 6. IEEE 2007. pp. 53-59.

225. Lindholm, T., Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley. 1999.

226. Lindfors, J., Fleury, M. JMX: Managing J2EE with Java Management Extensions. SAMS 2002.

227. Little, M. C., Wheater, S. M. *Building Configurable Applications in Java*. Proc. of Intl. Conference on Configurable Distributed Systems. INSPEC 1998. pp. 172-179.

228. Lopes, R., F., Silva, F. J. *Strong Migration in a Grid based on Mobile Agents.* Transactions on Systems. Vol. 4, Nr. 10. WSEAS 2005. pp. 1687-1694.

229. Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., Hofman, R. *Efficient Java RMI for Parallel Programming*. Transactions on Programming Languages and System. Vol. 23, Nr. 6. ACM 2001. pp. 747-775.

230. Makimoto, T., Eguchi, K., Yoneyama, M. *The Cooler The Better: New Directions in Nomadic Age*. IEEE Computer. Vol. 34, Nr. 4. IEEE 2001. pp. 38-42.

231. Manoel, E., Brumfield, S. C., Converse, M., DuMont, M., Hand, L., Lilly, G., Moeller, M., Nemati, A., Waisanen, Al. *Provisioning On Demand: Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*. IBM Redbook 2003.

232. Marvic, R., Merle, P., Geib, J.-M. *Towards a Dynamic CORBA Component Platform*. Proc. of the 2nd Intl. Symposium on Distributed Objects and Applications (DOA 2000). IEEE 2000. pp. 305-314.

233. McKinley, P. K, Sadjadi, S. M., Cheng, B. H. C. *Composing Adaptive Software.* IEEE Computer. Vol. 37, Nr. 7. IEEE 2004. pp. 56-64.

234. McLean, S., Williams, K., Naftel, J. *Microsoft .NET Remoting*. Microsoft Press. 2002.

235. *MEDIAGRID*
http://www.d-grid-ggmbh.de/index.php?id=112

236. Merino, L. R., Lopez, L., Fernandez, A., Cholvi, V. *DANTE: A Self-Adapting Peer-to-Peer System*. Proc. of the Intl. Conference on Agent and P2P Computing. LNAI 4461. Springer 2008. pp. 31-42.

237. Meyer, B. *.NET is coming*. IEEE Computer. Vol. 34, Nr. 8. IEEE 2001. pp. 92-97.

238. *Microsoft Silverlight*
http://www.microsoft.com/silverlight/

239. *Microsoft Windows Registry*
http://en.wikipedia.org/wiki/Windows_Registry/

240. Milenkovic, M., Robinson, S. H., Knauerhase, R. C., Barkai, D., Garg, S., Tewari, V., Anderson, T. A., Bowman, M. *Toward Internet Distributed Computing*. IEEE Computer. Vol. 36, Nr. 5. IEEE 2003. pp. 38-46.

241. Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S. *Process Migration*. ACM Computing Surveys. Vol. 32, Nr. 3. ACM 2000. pp. 241-299.

242. Monson-Haefel, R. *Enterprise JavaBeans*. 2nd edition. O'Reilly 2000.

243. Montanari, R., Tonti, G., Stefanelli, C. *Policy-Based Separation of Concerns for Dynamic Code Mobility Management*. Proc. of the 27th Intl. Conference on Computer Software and Applications. IEEE 2003. pp. 82-90.

244. Montanari, R., Lupu, E., Stefanelli, C. *Policy-Based Dynamic Reconfiguration of Mobile-Code Applications*. IEEE Computer. Vol. 37, Nr. 7. IEEE 2004. pp. 73-80.

245. Montresor, A. *A Robust Protocol for Building Superpeer Overlay Topologies*. Proc. of the 4th Intl. Conference on Peer-to-Peer Computing. IEEE 2004. pp. 202-209.

246. Moreau, L., Ribbens, D. *Mobile Objects in Java.* Scientific Programming. Vol. 10, Nr. 1. ACM 2002. pp. 91-100.

247. Mühlenschulte, Albrecht. *Lokalisierung von Ressourcen in einem P2P Netzwerk*. Student Thesis. University of Marburg 2006.

248. Müller, H. A., Kienle, H. M., Stege, U. *Autonomic Computing Now You See It, Now You Don't. Design and Evolution of Autonomic Software Systems*. Proc. of the Intl. Summer Schools on Software Engineering. LNCS 5413. Springer 2009. pp. 32-54.

249. Munch-Ellingsen, A., Eriksen, D. P., Andersen, A. *Argos, an Extensible Personal Application Server*. Prof. of the Intl. Conference on Middleware 2007. LNCS 4834. Springer 2007. pp. 21-40.

250. Murch, R. *Autonomic Computing*. IBM Press 2004.

251. Myers, B. A. *Using Hand-Held Devices and PCs Together*. Communications of the ACM. ACM 2001. pp. 34-41.

252. Narasimhan, P., Moser, L. E., Melliar-Smith, P. M. *Using Interceptors to Enhance CORBA*. IEEE Computer. Vol. 32, Nr. 7. IEEE 1999. pp. 62-68.

253. Neable, C. *The .NET Compact Framework*. IEEE Pervasive Computing. Vol. 1, Nr. 4. IEEE 2002. pp. 84-87.

254. *Netx*
http://jnlp.sourceforge.net/netx/compare.html.

255. Nichols, D. *Using Idle Workstations in a Shared Computing Environment*. ACM SIGOPS Operating Systems Review. Vol. 21, Nr. 5. ACM 1987, pp. 5-12.

256. *NOMADS*
http://www.coginst.uwf.edu/projects/nomads/index.html

257. *Novell Mono*
http://www.mono-project.com

258. Oaks, S. Wong, H. *Java Threads*. O'Reilly & Assoc. 1997.

259. OASIS. *Web Services Resource Framework (WSRF)*
http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-01.pdf

260. *Object Component Desktop*
http://ocd.sourceforge.net/docs/index.html.

261. *Open Grid Service Architecture*
http://www.globus.org/ogsa/

262. *OSGi The Dynamic Module System for Java.* OSGi Alliance 2009
http://osgi.org.

263. Orfali, R., Harkey, D. *Client/Server Programming with Java and CORBA*. 2nd edition. John Wiley & Sons, Inc. 1998.

264. O'Sullivan, D., Wade, V., Lewis, D. *Understanding as We Roam*. IEEE Internet Computing. Vol. 11, Nr. 2. IEEE 2007. pp. 26-33.

265. Paal, S., Kammüller, R., Freisleben, B. *Distributed Extension of Internet Information Systems.* Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2001). Anaheim, USA. IASTED 2001. pp. 38-43.

266. Paal, S., Kammüller, R., Freisleben, B. *Dynamic Composition of Web Server Functionality over the Internet*. Proceedings of the 6th Intl. WebNet World Conference of the WWW, Internet, and Intranet (Webnet 2001). Orlando, USA. AACE 2001. pp. 967-972.

267. Paal, S., Kammüller, R., Freisleben, B. *Java Class Separation for Multi-Application Hosting.* Proceedings of the 3rd Intl. Conference on Internet Computing (IC 2002). Las Vegas, USA. CSREA 2002. pp. 259-266.

268. Paal, S., Kammüller, R., Freisleben, B. *Java Class Deployment with Class Collections*. Proceedings of the 3rd Intl. Conference on Objects, Components, Architectures, Services and Applications for a Networked World (NODE 2002). Erfurt, Germany. 2002. pp. 144-158.

269. Paal, S., Kammüller, R., Freisleben, B. *Customizable Deployment, Composition and Hosting of Distributed Java Applications*. Proceedings of the 3rd Intl. Conference on Distributed Objects and Applications (DOA 2002). LNCS 2519. Irvine, USA. Springer 2002. pp. 845-865.

270. Paal, S., Kammüller, R., Freisleben, B. *Java Class Deployment with Class Collections*. Objects, Components, Architectures, Services, and Applications for a NetworkedWorld. LNCS 2591. Erfurt, Germany. Springer 2003. pp. 135-151.

271. Paal, S., Kammüller, R., Freisleben, B. *Java Remote Object Binding with Method Streaming*. Proceedings of the 4th Intl. Conference on Objects, Components, Architectures, Services and Applications for a Networked World (NODE 2003). Erfurt, Germany, 2003. pp. 230-244.

272. Paal, S., Kammüller, R., Freisleben, B. *Separating the Concerns of Distributed Deployment and Dynamic Composition in Internet Application Systems*. Proceedings of the 4th Intl. Conference on Distributed Objects and Applications (DOA 2003). LNCS 2888. Catania, Italy. Springer 2003. pp. 1292-1311.

273. Paal, S., Kammüller, R., Freisleben, B. *Self-Managing Remote Object Interconnection*. Proceedings of the 15th Intl. Conference and Workshop on Database and Expert Systems Applications (DEXA 2004). Zaragoza, Spain. IEEE 2004. pp. 758-763.

274. Paal, S., Kammüller, R., Freisleben, B. *A Cross-Platform Application Environment for Nomadic Desktop Computing*. Proceedings of the Intl. Conference on Objects, Components, Architectures, Services, and Applications for a NetworkedWorld (NODE 2004). LNCS 3263. Erfurt, Germany. Springer 2004. pp. 185-200.

275. Paal, S., Kammüller, R., Freisleben, B. *Supporting Nomadic Desktop Computing using an Internet Application Workbench*. Proceedings of the 5th Intl. Conference and Workshop on Distributed Objects and Applications (DOA 2004). Larnaca, Cyprus. Springer 2004. pp. 40-43.

276. Paal, S., Novak, J., Freisleben, B. *Kollektives Wissensmanagement in virtuellen Gemeinschaften*. Wissensprozesse in der Netzwerkgesellschaft. transcript Verlag 2004. pp. 119-143.

277. Paal, S., Kammüller, R., Freisleben, B. *Dynamic Software Deployment with Distributed Application Repositories*. 14. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005). Informatik aktuell. Kaiserlautern, Germany. Springer 2005. pp. 41-52.

278. Paal, S., Kammüller, R., Freisleben, B. *Application Object Isolation in Cross-Platform Operating Environments*. Proceedings of the 6th Intl. Symposium on Distributed Objects and Applications (DOA 2005). LNCS 3761. Agia Napa, Cyprus. Springer 2005. pp. 1047-1064.

279. Paal, S., Kammüller, R., Freisleben, B. *An Autonomic Cross-Platform Operating Environment for On Demand Internet Computing*. Demonstration on the 6th International Middleware Conference (MW 2005). Grenoble, France. 2005.

280. Paal, S., Kammüller, R., Freisleben, B. *Crosslets: Self-Managing Application Deployment in a Cross-Platform Operating Environment*. Proceedings of the 3rd Intl. Conference on Component Deployment (CD 2005). LNCS 3798. Grenoble, France. Springer 2005. pp. 51-65.

281. Paal, S., Kammüller, R., Freisleben, B. *Crossware: Integration Middleware for Autonomic Cross-Platform Internet Application Environments*. Journal on Integrated Computer-Aided Engineering. Vol. 13, Nr. 1. IOS Press 2006. pp. 41-62.

282. Paal, S., Kammüller, R., Freisleben, B. *Self-Managing Application Composition for Cross-Platform Operating Environments*. Proceedings of the 2nd IEEE Intl. Conference on Autonomic and Autonomous Systems (ICAS 2006). Silicon Valley, USA. IEEE 2006. p. 37.

283. Paal, S., Bröcker, L., Borowski, M. *Supporting On-Demand Collaboration in Web-Based Communities*. Proceedings of the 17th IEEE Intl. Conference on Database and Expert Systems Applications (DEXA 2006). Krakow, Poland. IEEE 2006. pp. 293-298.

284. Paal, S. *ODIX: An On-Demand Internet Application Workbench*. Proceedings of the 9th Intl. Conference on Internet Computing (ICOMP 2008). Las Vegas, USA. CSREA 2008. pp. 342-348.

285. Pairot, C., Garcia, P., Mondejar, R., Skarmeta, A. F. *Building Wide-Area Collaborative Applications on Top of Structured Peer-to-Peer Overlays*. Proc. of the 14th Intl. Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises. IEEE 2005. pp. 350-355.

286. Papazoglou, M.P., Georgakopoulos, D. *Service Oriented Computing*. Communications of ACM. Vol. 46, Nr. 10. ACM 2003. pp. 24-28.

287. Parameswaran, M., Susarla, A., Whinston, A. B. *P2P Networking: An Information-Sharing Alternative*. IEEE Computer. Vol. 34, Nr. 7. IEEE 2001. pp. 31-38.

288. Parker, D., Cleary, D. *A P2P Approach to ClassLoading in Java*. Proc of the Intl. Conference on Agents and Peer-to-Peer Computing. LNAI 2872. Springer 2003. pp. 144-149.

289. Parkin, M., Brooke, J. M. *A PDA Client for the Computational Grid*. Concurrency: Practice and Experiences. Vol. 19, Nr. 9. John Wiley & Sons, Ltd. 2006. pp. 1317-1331.

290. Parlavantzas, C.G., Blair, G. *An Extensible Binding Framework for Component-Based Middleware*. Proc. of the Intl. Conference on Enterprise Distributed Objects Computing. IEEE 2003. pp. 252.

291. Pashtan, A., Heusser, A., Scheuermann, P. *Personal Service Areas for Mobile Web Applications*. IEEE Internet Computing. IEEE 2004. Vol. 8, Nr. 6. pp. 34-39.

292. Paton, N. W., de Aragao, M. A. T., Lee, K., Fernandes, A., Sakellariou, R. *Optimizing Utility in Cloud Computing through Autonomic Workload Execution*. IEEE Bulletin of the Technical Committee on Data Engineering. Vol. 32, Nr. 1. IEEE 2009. pp. 51-58.

293. Paulson, L. D. *Building Rich Web Applications with AJAX*. IEEE Computer. Vol. 38, Nr. 10. IEEE 2005. pp. 14-17.

294. Picco, G. P. *Mobile Agents: An Introduction*. Journal on Microprocessors and Microsystems. Vol. 25, Nr. 2. Elsevier 2001. pp. 65-74.

295. Ponzo, J., Hasson, L. D., George, J., Thomas, G., Gruber O. et. al. *On Demand Web-Client Technologies*. IBM Systems Journal. Vol. 43, Nr. 2. IBM 2004. pp. 297-315.

296. *PowerUpdate*
http://www.zerog.com/products_pu.html.

297. Psotta, R. *Automatische Konfiguration von heterogenen Internet-Anwendungssystemen für plattformtransparente Benutzerumgebungen*. Diploma Thesis. University of Marburg 2004.

298. Puder, A. *A Code Migration Framework for AJAX Applications*. Proc. of the Intl. Conference on Distributed Applications and Interoperable Systems. LNCS 4025. Springer 2006. pp. 138-151.

299. Qi, R., Wang, Z., Li, S. *JMobile: A Lighweight Transparent Migration Mechanism for Mobile Agents*. Proc. of the Intl. Conference on Wireless Communications, Networking and Mobile Computing (WiCOM 2008). IEEE 2008. pp. 1-4.

300. Richardson, T., Stafford-Fraser, Q., Wood, K. R., Hopper, A. *Virtual Network Computing*. IEEE Internet Computing. Vol. 2, Nr. 1. IEEE 1998. pp. 33-38.

301. Richmond, M., Noble, J. *Reflections on Remote Reflecti*on. Proc. of the 24th Australasian Computer Science Conference (ACSC). IEEE 2001. pp. 163-170.

302. Rinat, R., Smith, S. *Modular Internet Programming with Cells*. Proc. of the ECOOP 2002. LNCS 2374. Springer 2002. pp. 257-280.

303. Ripeanu, M., Iamnitchi, A., Foster, I. *Mapping the Gnutella Network*. IEEE Internet Computing. Vol. 6, Nr. 1. IEEE 2002. pp. 50-57.

304. Roberts, L. G. *Beyond Moore's Law: Internet Growth Trends*. IEEE Computer. Vol. 33, Nr. 1. IEEE 2000. pp. 117-119.

305. Roscoe, T., Lyles, B. *Distributing Processing without DPEs: Design Considerations for Public Computing Platforms*. Proc. of the ACM SIGOPS European Workshop: Beyond the PC - New Challenges for the Operating System. ACM 2000. pp. 235-240.

306. Rowstron, A., Druschel, P. *Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems*. Proc. of the Intl. Conference on Middleware (MW 2001). LNCS 3347. Springer 2001. pp. 329-350.

307. Royon, Y., Frenot, S., Le Mouel, F. *Virtualization of Service Gateways in Multi-Provider Environments*. Proc. of the Intl. Conference on Component Based Software Engineering. LNCS 4063. Springer 2006. pp. 385-392.

308. Rubin, A. D., Geer, D. E. *Mobile Code Security*. IEEE Internet Computing. Vol. 2, Nr. 6. IEEE 1998. pp. 30-34.

309. Ruth, P., Jiang, X., Xu, D., Goasguen, S. *Virtual Distributed Environments in a Shared Infrastructure*. IEEE Computer. Vol. 38, Nr. 5. IEEE 2005. pp. 63-69.

310. Sadjadi, S. M., McKinley, P. K., Cheng, B. H. C., Stirewalt, R. E. *TRAP/J: Transparent Generation of Adaptable Java Programs.* Proc. of the 6th Intl. Symposium on Distributed Objects and Applications (DOA). LNCS 3291. Springer 2004. pp. 1243-1261.

311. Sakamoto, T., Sekiguchi, T., Yonezawa, A. *Bytecode Transformation for Portable Thread Migration in Java.* Proc. of the Joint Symposium on Agent Systems, Mobile Agents, and Applications. LNCS 1882. Springer 2000. pp. 443-481.

312. Santos, N., Marques, P., Silva, L. M., Silva, J. G. *A Framework for Smart Proxies and Interceptors in RMI*. Proc. of the Intl. Conference on Parallel and Distributed Computing Systems. IASTED 2002. pp. 1-18.

313. Satoh, I. *Self-Deployment of Distributed Applications*. Proc. of the Intl. Conference on Scientific Engineering of Distributed Java Applications (FIDJI 2004). LNCS 3409. Springer 2004. pp. 48-57.

314. Satyanarayanan, M. *Pervasive Computing: Vision and Challenges*. Personal Communication. IEEE 2001. pp. 10-17.

315. Satyanarayanan, M., Gilbert, B., Toups, M., Tolia, N., Surie, A., O’Hallaron, D. R., Wolbach, A., Harkes, J., Perrig, A., Farber, D. J., Kozuch, M. A., Helfrich, C. J., Nath, P., Lagar-Cavilla, H.-A. *Pervasive Personal Computing in an Internet Suspend/Resume System*. IEEE Internet Computing. Vol. 11, Nr. 2. IEEE 2007. pp. 16-25.

316. Satyanarayanan, M., Kozuch, M. A., Helfrich, C. J., O'Halleron, D. R. *Towards Seamless Mobility on Pervasive Hardware*. Pervasive and Mobile Computing, Vol. 1, Nr. 2. Elsevier 2005. pp. 157-189.

317. Schmeck, H. *Organic Computing - A New Vision for Distributed Embedded Systems*. Proc. of the Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005). IEEE 2005. pp. 201-203.

318. Schmidt, K. *Verteilte On-Demand Prozessierung von Multimediadaten*. Diploma Thesis. University of Siegen 2010.

319. Schulze, B., Madeira, E. *Migration Transparency in Agent Systems.* IEICE/IEEE Special Issue on Autonomous Decentralized Systems. Vol. E83-B, Nr. 5. IEEE 2000. pp. 942-950.

320. Seacord, R. C., Hissam, S. A., Wallnau, K. C. *AGORA: A Search Engine for Software Components*. IEEE Internet Computing. Vol. 2, Nr. 6. IEEE 1998. pp. 62-70.

321. *Secure and Open Mobile Agent*
http://lia.deis.unibo.it/Research/SOMA/

322. Seely, S. *SOAP: Cross Platform Web Service Development Using XML*. Prentice Hall. Hartford, Wisconsin. 2001.

323. Sharma, V. S., Jalote, P. *Deploying Software Components for Performance*. Proc. of the Intl. Conference on Component Based Software Engineering. LNCS 5282. Springer 2008. pp. 32-47.

324. Silver, N. *Jtrix: Web Services beyond SOAP*. JavaWorld. IDG 2002. Nr. 5. http://www.javaworld.com/javaworld/jw-05-2002/jw-0503-jtrix_p.html

325. Sinner, A., Kleemann, T., von Hessling, A. *Semantic User Profiles and their Applications in a Mobile Environment.* Proc. of Artificial Intelligence in Mobile Systems (AIMS). 2004.

326. Smith, M., Friese, T., Freisleben, B. *Towards a Service-Oriented Ad Hoc Grid*. Proc. of the Intl. Symposium on Parallel and Distributed Computing. IEEE 2004. pp. 201-208.

327. Smith, D. E., Nair, R. *The Architecture of Virtual Machines*. IEEE Computer. Vol. 38, Nr. 5. IEEE 2005. pp. 32-38.

328. Sotomayor, B., Montero, R. S., Foster, I. *Virtual Infrastructure Management in Private and Hybrid Clouds*. IEEE Internet Computing. Vol. 13, Nr. 5. IEEE 2009. pp. 14-22.

329. Spotswood, M. *System and Method for using a Classloader Hierarchy to Load Software Applications*. United States Patent 2004/0255293 A1. 2004.

330. *Spring Framework*
http://www.springsource.com.

331. Srinivas, R. N. *Java Web Start to the Rescue.* JavaWorld. IDG 2001. Nr. 7 .
http://www.javaworld.com/javaworld/jw-07-2001/jw-0706-webstart_p.html.

332. van Steen, M., Homburg, P., Tanenbaum, A. S. *Globe: A Wide-Area Distributed System*. IEEE Concurrency. Vol. 7, Nr. 1. IEEE 1999. pp. 70-78.

333. di Stefano, A., Santoro, C. *NetChaser: Agent Support for Personal Mobility*. IEEE Internet Computing. Vol. 4, Nr. 2. IEEE 2000. pp. 74-79.

334. Stoica, I. Morris, R., Karger, D., Kaashoek, M. Balakrishnan, H. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. ACM 2001. pp. 149-160.

335. Sudmann, N. P., Johansen, D. *Software Deployment Using Mobile Agents*. Proc. of the Intl. Conference on Component Deployment (CD 2002). LNCS 2370. Springer 2002. pp. 97-107.

336. *Sun Enterprise Java Beans Technology*
http://java.sun.com/products/ejb/

337. *Sun Java Class Data Sharing*
http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html

338. Sun Java Community Process. *JSR-121: Application Isolation API Specification*.
http://jcp.org/jsr/detail/121.jsp

339. Sun *Java Dynamic Proxy Classes*
http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

340. Sun *Java Thread Primitive Deprecation*
http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

341. *Sun JavaFX*
http://java.sun.com/javafx/

342. *Sun Java Management Extensions (JMX)*
http://java.sun.com/products/JavaManagement/

343. *Sun Java Naming and Directory Interface (JNDI)*
http://java.sun.com/products/jndi/

344. *Sun Java Preferences API*
http://java.sun.com/javase/6/docs/technotes/guides/preferences/

345. *Sun Java Server Faces*
https://javaserverfaces.dev.java.net/

346. *Sun Java Store*
http://java.com/en/store/

347. *Sun Jini*
http://www.jini.org

348. Suri, N., Bradshaw, J., Breedy, M., Groth, P., Hill, G., Jeffers, R., and Mitrovich, T. *An Overview of the NOMADS Mobile Agent System*. Proc. of the 2nd Intl. Symposium on Agent Systems and Applications. ACM 2000. pp. 94-100.

349. Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., Jeffers, R. *Strong Mobility and Fine-Grained Resource Control in NOMADS*. Proc. of the Joint Symposium on Agent Systems, Mobile Agents, and Applications. LNCS 1882. Springer 2000. pp. 2-15.

350. Talia, D. *The Open Grid Service Architecture - Where the Grid Meets the Web*. IEEE Internet Computing. Vol. 6, Nr. 6. IEEE 2002. pp. 67-71.

351. Talwar, V., Wu, Q., Pu, C., Yan, W., Jung, G., Milojicic, D. *Comparison of Approaches to Service Deployment*. Proc. of the Intl. Conference on Distributed Computing Systems. (ICSCS). IEEE 2005. pp. 543-552.

352. Tate, B. *Better, Faster, Lighter Java*. O'Reilly. 2004.

353. *TeamDrive*
http://www.teamdrive.net/

354. Thain, D., Tannenbaum, T., Livny, M. *Distributed Computing in Practice: The Condor Experience*. Concurrency and Computation: Practice and Experience. Vol. 17, No. 2-4. Wiley 2005. pp. 323-356.

355. Thomas, R. *Autonomic Software Deployment in verteilten Internet-Anwendungssystemen.* Diploma Thesis. University of Marburg 2007.

356. Tilevich, E., Smaragdakis, Y. *J-Orchestra: Automatic Java Application Partitioning.* Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP). LNCS 2374. Malaga, Spain. Springer 2002. pp. 178-204.

357. Trautweiler, M. *Transparent Runtime Evolution of Components*. Master Thesis. ETH Zürich 2004.

358. Truyen, E. Robben, B., Vanhaute, B., Coninx, T., Joosen, W., Verbaeten, P. *Portable Support for Transparent Thread Migration in Java*. Proc. of the Joint Symposium on Agent Systems, Mobile Agents, and Applications. LNCS 1882. Springer 2000. pp. 377-426.

359. Truman, T. E., Pering, T., Doering, R., Brodersen, R. W. *The InfoPad Multimedia Terminal: A Portable Device for Wireless Information Access*. IEEE Transaction on Computers. Vol. 47, Nr. 10. IEEE 1998. pp. 1073-1087.

360. Unger, T. *Dynamische Verteilung und Komposition von Internet Anwendungen*. Diploma Thesis. University of Siegen 2004.

361. *UNICORE - Uniform Interface to Computing Resources* http://www.unicore.eu/

362. Utsch, G. *Aufbau einer erweiterbaren Benutzerverwaltung und Softwarekonfiguration über Verzeichnisdienste in Java*. Diploma Thesis. University of Siegen 1998.

363. Vahdat, A., Anderson, T., Dahlin, M., Culler, D., Belani, E., Eastham, P., Yoshikawa, C. *WebOS: Operating System Services for Wide Area Applications*. Intl. Symposium on High Performance Distributed Computing. IEEE 1998. pp. 52-63.

364. Vallecillo, A. *RM-ODP: The ISO Reference Model for Open Distributed Processing*. DINTEL Edition on Software Engineering. 2001. Nr. 3. pp. 69-99.

365. Vaughan-Nichols, S. J. *Developing the Distributed Computing OS*. IEEE Computer. Vol. 35, Nr. 9. IEEE 2002. pp. 19-21.

366. Vaughan-Nichols, S. *Web Services: Beyond the Hype*. IEEE Computer. Vol. 35, Nr. 2. IEEE 2002. pp. 18-21.

367. Venners, B. *Inside The Java 2 Virtual Machine*. McGraw-Hill 1999.

368. Venkitachalam, G., Chiueh, T. *High Performance Common Gateway Interface Invocation*. Proc. of the Intl. Workshop on Internet Applications. IEEE 1999. pp. 4-11.

369. Viega, J. *Cloud Computing and the Common Man*. IEEE Computer. Vol. 42, Nr. 8. IEEE 2009. pp. 106-108.

370. Vinoski, S. *CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments*. IEEE Communications. Vol. 35, Nr. 2. IEEE 1997. pp. 46-55.

371. Vinoski, S. *Chain of Responsibility*. IEEE Internet Computing. Vol. 6, Nr. 6. IEEE 2002. pp. 80-83.

372. Vouk, M. A. *Cloud Computing – Issues, Research and Implementations*. Proc. of the Intl. Conference on Information Technology Interfaces. ITI 2008. pp. 31-40.

373. Waldo, J. *Alive and Well: Jini Technology Today*. IEEE Computer. Vol. 33, Nr. 6. IEEE 2000. pp. 107-109.

374. Waldo, J. *The Jini Architecture for Network-Centric Computing*. Communications of the ACM. Vol. 42, Nr. 7. ACM 1999. pp. 76-82.

375. Walsh, W. E., Tesauro, G., Kephart, J. O., Das, R. *Utility Functions in Autonomic Computing*. Proc. of the Intl. Conference on Autonomic Computing. IEEE 2004. pp. 70-77.

376. Want, R., Perint, T., Tennenhouse, D. *Comparing Autonomic and Proactive Computing*. IBM Systems Journal. Vol. 42, Nr. 1. IBM 2003. pp. 129-135.

377. Watson, M. *Sun One Services (Professional Middleware)*. Hungry Minds. 2002.

378. Wege, C. *Portal Server Technology*. IEEE Internet Computing. Vol. 6, Nr. 3. IEEE 2002. pp. 73-77.

379. Weiser, M. *Ubiquitous Computing.* IEEE Computer. Vol. 26, Nr. 10. IEEE 1993. pp. 71-72.

380. Weiser, M. *The Computer for the 21st Century*. IEEE Pervasive Computing. Vol. 1, Nr. 1. IEEE 2002. pp. 19-25.

381. White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., Kephart, J. O. *An Architectural Approach to Autonomic Computing*. Proc. of the Intl. Conference on Autonomic Computing (ICAC 2004). IEEE 2004. pp. 2-9.

382. *WIKINGER.*
http://www.wikinger-escience.de

383. Wikipedia. *Sun Java Web Start*
http://de.wikipedia.org/wiki/Java_Web_Start.

384. Wikipedia. *Roaming User Profile*
http://en.wikipedia.org/wiki/Roaming_user_profile.

385. Wikipedia. *Virtual Machine*.
http://en.wikipedia.org/wiki/Virtual_machine

386. Wojciechowski, P.T., Sewell, P. *Nomadic Pict: Language and Infrastructure Design for Mobile Agents*. IEEE Concurrency. Vol. 8, Nr. 2. IEEE 2000. pp. 42-52.

387. Wood, K. R. et al. *Global Teleporting with Java: Toward Ubiquitous Personalized Computing.* IEEE Computer. Vol. 30, No. 2. IEEE 1997. pp. 53-59.

388. Wright, J. M., Dietrich, J. B. *Requirements for Rich Internet Application Design Methodologies*. Proc. of the Intl. Conference on Web Information System Engineering. LNCS 5175. Springer 2008. pp. 106-119.

389. *X Window System*
http://www.x.org

390. *Xito AppManager*
http://xito.sourceforge.net/

391. Zachariadis, S. Mascolo, C., Emmerich W. *SATIN: A Component Model for Mobile Self-Organisation.* Proc. of the 5th Intl. Conference on Distributed Applications (DOA 2004). LNCS 3291. Agia, Napa, Cyprus. Springer 2004. pp. 1303-1321.

392. Zachary, J. *Protecting Mobile Code in the Wild*. IEEE Internet Computing. Vol. 7, Nr. 2. IEEE 2003. pp. 78-82.

393. Zhang, L., Ardagna, D. *SLA Based Profit Optimization in Autonomic Computing Systems*. Proc. of the Intl. Conference on Service Oriented Computing. ACM 2004. pp. 173-182.

394. Zhao, B. Y., Kubiatowicz, J. D., Joseph, A. D. *Tapestry: An Infrastructure for Fault Resilient Wide-Area Location and Routing*. TR UCB//CSD-01-1141. U. C. Berkeley 2001.

395. Zhu, J., Törö, M., Leung, V., Vuong, S. *Supporting Universal Personal Computing on Internet with Java and CORBA*. Proc. of the Intl. Workshop on Java for High-Performance Network Computing. ACM 1998. pp. 1007-1013.

396. Zhu, W., Wang, C. L., Lau, F. *JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*. Proc. of the 4th Intl. Conference on Cluster Computing, IEEE 2002. pp. 381-388.

397. Zhu, W., Wang, C. L., Lau, F. *Lightweight Transparent Java Thread Migration for Distributed JVM*. Proc. of the Intl. Conference on Parallel Processing. IEEE 2003. pp. 465-472.

398. Zhou, Y., Zhao, Q., Perry, M. *Reasoning over Ontologies of On Demand Service*. Proc. of the Intl. Conference on e-Technology, e-Commerce and e-Service. IEEE 2005. pp. 381-384.

Erklärung

Ich versichere, dass ich meine Dissertation

"An Autonomic Cross-Platform Operating Environment
for On-Demand Internet Computing"

selbständig, ohne unerlaubte Hilfe angefertigt und mich dabei keiner anderen als der von mir ausdrücklich bezeichneten Quellen und Hilfen bedient habe.

Die Dissertation wurde in der jetzigen oder einer ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

(Ort / Datum)	(Unterschrift mit Vor- und Zuname)

Curriculum Vitae

Stefan Paal
Saarstrasse 1a

50677 Köln

Geburtsdatum	26.05.1969
Geburtsort	Germersheim/Rheinland-Pfalz
Familienstand	ledig
Staatsangehörigkeit	deutsch

August 1975 - Juni 1985	**Mittlere Reife** Grundschule Lustadt, Gymnasium Germersheim
August 1985 - Juli 1988	**Informationselektroniker** Berufsausbildung, BASF AG Ludwigshafen
August 1988 - Juni 1989	**Fachhochschulreife** Fachoberschule Elektrotechnik Neustadt/Weinstrasse
Oktober 1990 - Oktober 1995	**Dipl.-Ing. Elektrotechnik** Universität Siegen, Studium der Elektrotechnik
Dezember 1995 - November 2000	**Wissenschaftlicher Mitarbeiter** Universität Siegen
Seit Dezember 2000	**Wissenschaftlicher Mitarbeiter** Fraunhofer Gesellschaft, Sankt Augustin

Köln, den 28. April 2010

Stefan Paal