Plug-and-Participate for Limited Devices in the Field of Industrial Automation

Dissertation
zur

Erlangung des Doktorgrades
der Naturwissenschaften
(Dr. rer. nat.)

dem

Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg

vorgelegt von

Steffen Deter
aus Wippra

Marburg/Lahn 2003

Vom Fachbereich Mathematik und Informatik

der Philipps-Universität Marburg

als Dissertation am _____ angenommen.

Erstgutachter: Prof. Dr. Manfred Sommer

Zweitgutachter: Prof. Dr. Bernd Freisleben

Tag der mündlichen Prüfung: _____

Danksagung

Mit dieser Danksagung möchte ich all diejenigen ansprechen, die direkt oder indirekt mit zu dieser Arbeit beigetragen haben. Dies sind zunächst meine Eltern, die mir mein Studium ermöglicht haben. Dank gilt insbesondere meinem Doktorvater Herrn Prof. Dr. Manfred Sommer für seine Betreuung der Arbeit, für seine Ratschläge, Hinweise und Kritiken zur Arbeit, aber auch für die Bemühungen um meine Einstellung und Weiterbeschäftigung im PABADIS Projekt. Gleichsam möchte ich mich auch bei meinem Zweitgutachter Herrn Prof. Dr. Bernd Freisleben für die Hinweise und Anregungen bedanken, die zur weiteren Verbesserung der Arbeit führten.

Ganz besonderes möchte ich meinen Kollegen Jörn Peschke von der Universität Magdeburg hervorheben: In unzähligen Diskussionen, Meetings und Telefonaten, sowie auf vielen gemeinsamen Dienstreisen haben wir Lösungen, Ansätze und Ideen für das PABADIS Projekt erörtert, die sowohl das Projekt an sich, und damit auch die hier vorliegende Dissertation nachhaltig geprägt haben. Diese Zusammenarbeit betrifft auch die Kollegen Dr. Axel Klostermeyer, Dr. Karsten Sohr, Herrn Markus Vincon sowie das PABADIS Team - sie halfen mir als Ratgeber, Ansprechpartner, Kollegen, und vor allem als gute Freunde. Dr. Arndt Lüder und Dr. Axel Schröder, "strenge Kritiker" der Arbeit, gaben wertvolle Hinweise und Anregungen, um die Arbeit wesentlich zu verbessern. Schließlich will ich aus dem Kreise der Kollegen auch meine Büronachbarin Frau Barbara Krzensk nennen - die netten und unterhaltsamen Gespräche haben das Arbeiten oftmals bunter gestaltet. Gleiches gilt für meine Freundin Yanti, die es immer wieder verstand, mich der Welt der Informatik und Automatisierung zu entreissen. Sicherlich, sie hatte es nicht immer leicht, ein "Arbeitspferd" wie mich zu zügeln. Sie hat zudem zusammen mit Herrn Nicholas Stearn geholfen, die sprachliche Seite der Arbeit zu überprüfen. Speziell Herr Stearn war als Englisch-Muttersprachler eine große Hilfe.

Last but not least bin ich der Sekretärin Frau Lydia Heinbächer zu Dank verpflichtet für die Ausrichtung und Vorbereitung von Meetings und die unermüdliche Bearbeitung meiner (zahlreichen) Reisekostenabrechnungen.

Kurzfassung

Ausgangspunkt und gleichzeitig Motivation dieser Arbeit ist die heutige Marktsituation: Starke Kundenbedürfnisse nach individuellen Gütern stehen oftmals eher auf Massenproduktion ausgerichteten Planungs- und Automatisierungssystemen gegenüber - die Befriedigung individueller Kundenbedürfnisse setzt aber Flexibilität und Anpassungsfähigkeit voraus. Ziel dieser Arbeit ist es daher, einen Beitrag zu leisten, der es Unternehmen ermöglichen soll, auf diese individuellen Bedürfnisse flexibel reagieren zu können. Hierbei kann es im Rahmen der Dissertation natürlich nicht um eine Revolutionierung der gesamten Automatisierungs- und Planungslandschaft gehen; vielmehr ist die Lösung, die der Autor der Arbeit präsentiert, ein integraler Bestandteil eines Automatisierungskonzeptes, das im Rahmen des PABADIS Projektes entwickelt wurde: Während PABADIS das gesamte Spektrum von Planung und Maschineninfrastruktur zum Inhalt hat, bezieht sich der Kern dieser Arbeit weitestgehend auf den letztgenannten Punkt - Maschineninfrastruktur. Ziel war es, generische Maschinenfunktionalität in einem Netzwerk anzubieten, durch das Fertigungsaufträge selbstständig navigieren.

Als Lösung präsentiert diese Dissertation ein Plug-and-Participate basiertes Konzept, welches beliebige Automatisierungsfunktionen in einer spontanen Gemeinschaft bereitstellt. Basis ist ein generisches Interface, in dem die generellen Anforderungen solcher ad-hoc Infrastrukturen aggregiert sind. Die Implementierung dieses Interfaces in der PABADIS Referenzimplementierung sowie die Gegenüberstellung der Systemanforderungen und Systemvoraussetzungen zeigte, das klassische Plug-and-Participate Technologien wie Jini und UPnP aufgrund ihrer Anforderungen nicht geeignet sind - Automatisierungsgeräte stellen oftmals nur eingeschränkte Ressourcen bereit. Daher wurde als zweites Ergebnis neben dem Plug-and-Participate basierten Automatisierungskonzept eine Plug-and-Participate Technologie entwickelt - Pini - die den Gegebenheiten der Automatisierungswelt gerecht wird und schließlich eine Anwendung von PABADIS auf heutigen Automatisierungsanlagen erlaubt. Grundlegende Konzepte von Pini, die dies ermöglichen, sind die gesamte Grundarchitektur auf Basis eines verteilten Lookup Service, die Art und Weise der Dienstrepräsentation sowie die effiziente Nutzung der angebotenen Dienste. Mit Pini und darauf aufbauenden Konzepten wie PLAP ist es nun insbesondere möglich, Automatisierungssysteme wie PABADIS auf heutigen Anlagen zu realisieren. Das wiederum ist ein Schritt in Richtung Kundenorientierung - solche Systeme sind mit Hinblick auf Flexibilität und Anpassungsfähigkeit gestaltet worden, um Kundenbedürfnissen effizient gerecht zu werden.

Contents

List of Figures

List of Code Examples

CHAPTER 1

Introduction and Motivation

"A successful manufacturing business is one which is able to respond rapidly to changes." (see [13], chapter 3, p. 5). This is a major manifest of business, but also in general: Being able to react appropriately to environmental changes - specifically in order to survive in such new environments - is the natural behavior of evolution. Research efforts in many fields have the objective to provide suitable mechanisms, behavior rules and patterns, algorithms, technologies, etc., that ensure the survival of the respective fields by following the evolutionary path.

For example, the field of plant automation has to define mechanisms and concepts that allow enterprises in the manufacturing business to follow suitable evolution paths, and thus to survive in the market. This claim of “allowing enterprises to survive in the market” by improving the internal structure, behavior and facilities of plants characterizes the field of plant automation as a subject of evolutionary processes.

Such evolutionary processes are related to the past 50 years in Germany, a time that was characterized by a change from seller’s markets to buyer’s markets (see [110], pp. 481). These buyer’s markets are mainly driven by customer behavior that can be abstracted by the Maslow pyramid (depicted in [64]) - customers intend to satisfy their individual demands, which in turn impact the behavior of sellers.

Recalling the first sentence of this thesis, enterprises must be able to react appropriately to such demands and trends. Reactions to changes in the enterprises’ environments, however, must be efficient in order to benefit from demands and trends (see also [84]). This requires a high degree of plant flexibility and adaptability. The basis lies in suitable plant infrastructures which first must be able to cope with a wide varity of products. A second aspect is the ability of a plant infrastructure to allow the

replacement of products in a production line by other products, depending on such demand(s). In the best case scenario, this replacement occurs immediately without much reconfiguration efforts of the manufacturing lines.

Remark: Such a replacement can be related to a complete product change, but also simply to a modification of a specific product feature. For example, a customer might order a black cabrio car, but a few days later he decides to alter the order to a red van - machines and planning mechanisms should be able to deal with such adapted orders.

The third aspect concerns the planning algorithms that must be able to manage changes, and the fourth is the ability of an enterprise to design the demanded products in rather short periods of time. Altogether, this results in the ability to bring new products or product variants quickly to the market in order to satisfy customer demands. An enterprise therefore can take advantage of a trend by shortening the time to market, and gaining higher margins than would be likely by a later entry to the market. The manifold consequences for enterprises can be summarized as follows:

- The development process must be faster and more efficient.
- The market analysis phase (i.e., obtaining information regarding the anticipated market chances) must be as short and precise as possible.
- The planning system of the enterprise/plant must be flexible and exact; it must be able to react efficiently and quickly.
- The production lines of the plant must be flexible and adaptable as well as suitable to produce different products without major changes in the system.

These listed consequences are not complete, but they represent the most important requirements. Taking a detailed look at them, the last two points mentioned can be identified as the key factors for a successful enterprise in turbulent market environments. Sucessfull enterprises, therefore, must:

- show a short time to market
- derive (early) benefits from trends
- (may) become a market leader because of its ability to be the first at the market

Further discussion take these basic requirements as starting point, and thus they also make up the main motivating focus of this work. Both - the requirements and the resulting consequences - raise several questions to be discussed:

- What do flexibility and adaptability mean regarding plants in general and, in particular, regarding production lines?
- What is the state of the art in plant systems and what kind of flexibility/adaptability such systems are able to provide?
- Which dimensions of plant automation systems are affected by the necessity to be flexible and adaptable?
- What are the processes to be considered in particular?

Although these questions will be discussed in depth in separate chapters (see chapters 2, 3 and 4), the most important arguments will be noted here, since they can be considered as the basic motivation for using plug-and-participate in plant automation.

The trend towards customization, large variation but smaller production lot size was already noted earlier in this chapter. Owing to this fact, production plants must be able to satisfy those widespread demands. A certain production capacity is required allowing different products and production steps to be processed on the same machine without major and expensive changes (pertaining to time and costs). If a plant structure, in contrast, provides a special machine for each production step, this strategy obviously does not pay off, since manufacturing machines are usually investment goods. If only small product lot sizes are produced on these machines, either the amortization period is rather long or the cost of the production steps performed at the machines are disproportionally high. Such unreasonable production costs will lead to expensive goods or quite possible to the insolvency of the enterprise. Plant infrastructures avoiding this by ensuring a certain flexibility and adaptability on the production-line level can be characterized as reconfigurable allowing the production of different products and variants of products on, e.g., multi-purpose machines.

This also has an impact on the planning systems of the plants because they also need to be redesigned - they must deal with flexible resources provided on the field level, and furthermore they must cope with turbulent market environments. Appropriate benefits will result from the planning system capability to combine flexible field-level resources efficiently as a reaction to demands, situations and trends. The resulting new planning paradigm necessarily must always consider the current plant configuration in order to dispatch production steps appropriately to available resources. However, today's planning systems are normally centralized systems (see the motivation chapters of [10, 45, 83]) using rather complex optimization algorithms in order to increase machine throughput and thus satisfy the market requirements. Such systems are generally designed to perform more or less long-term planning without consideration of possibly often-occurring new demands. The phrase "long-term planning" refers to the fact that once planning has been done, it can be seen as final and thus static - a change in the plant configuration or in demands may affect the entire planning. Such changes are difficult to implement; the most important aspect in this is that planning is rather time- and cost-intensive due to often-used complex optimization algorithms, and therefore those systems are not well-suited in turbulent environments. This orthogonal fashion of the advantages and disadvantages of centralized and distributed systems defines a trend towards distributed systems (see also [13, 68]) in order to provide a certain degree of flexibility and inrease the performance. Planning tasks will be processed in a distributed fashion - distributed across a network of different entities which generally is built on an appropriate plant network (field level). These tasks can be performed independently of each other, but also in cooperative way. Both facilities, independency and cooperation, allow systems to be easily able to react quickly and efficiently to changes in the plant configuration and the market environment.

This sequence of arguments, starting at the necessity of flexible resources and finally enforcing the definition of flexible planning algorithms based upon distributed systems, depicts a bottom-up direction of motivational reasons. However, the opposite direction, namely top-down starting with distributed planning systems, also defines requirements for the underlying field level. For example, the planning tasks independently dispatch manufacturing tasks to production units organized in a dedicated network. Such planning and dispatching processes require an always up-to-date overview of the resource network, where a certain fluctuation can be assumed: manufacturing units can freely join and leave the network, or can offer modified resources. Networks having those capabilities and features are commonly characterized by the term “dynamic networks” requiring specific technologies for network management. Among these basic facilities, planning systems also require efficient mechanisms to access the provided resources and to find these resources.

In comparing these requirements with modern technologies, this leads to plug-and-participate technologies that perfectly meet the sketched facets. They are suitable to create flexible network infrastructures of functionalities and capabilities rather than flexible device networks themselves. Such a functional view of the network is more abstract than a device-based view. Single nodes of such a cooperative network, either in the planning system or at field level essentially offer their functionality independently of the physical location instead of offering specific device functionality.

A simple example surveying this is a common flow through a plant, starting at the office level issuing an order: Such orders pass the planning system, and finally their respective tasks are dispatched to suitable field network entities. The planning system, especially, can be supported by plug-and-participate facilities during the planning and dispatching processes in order to be always up to date, and thus obtain a more optimal and reliable production cycle.

This simple example covered two particular dimensions of plant automation: the horizontal and the vertical dimensions. The horizontal dimension mainly concerns the field level, which should provide a homogeneous view of the upper layers of plant automation. Applying plug-and-participate mechanisms in conjuction with suitable interfaces, an abstraction of the field level can be achieved, which finally provides the most fundamental flexibility. Based upon this achieved flexibility, the vertical dimension of plant automation must be redesigned. This mainly concerns the mechanisms of the planning system - they must be adapted to the new paradigm in order be flexible. A more general view of all Manufacturing Execution System (MES) functionalities, such as resource allocation, scheduling and document control (for a list of distinct MES functionalities see also [54]) shows the necessity to redesign these functions in order to benefit from the horizontal integration. For example, the resource allocation and scheduling mechanisms are enabled to always use up-to-date information about the current plant state retrieved from an underlying plug-and-participate system. This allows for more precise/adequate planning of tasks issued by upper level systems, while the MES level can incorporate infrastructure changes immediately, especially if the planning is not done fully in advance. The most important point in this respect is that occuring infrastructure changes do not affect the entire planning, but only directly related tasks: Local infrastructure changes remain locally in terms of the re-planning of tasks. Such a flexible and efficient behavior of systems results from an independent definition of task paths through the plant - this independent path calculation keeps the planning processes concerning different tasks autonomous (except strictly necessary cooperation).

One plant automation solution covering the horizontal and vertical dimensions that aims at providing flexibility and adaptability is the PABADIS approach (Plant Automation Based on Distributed Systems, see also the PABADIS home page [77]): This approach constitutes a plant automation solution for an alternative mechanism for task assignment to available resources, which fundamentally relies on the actual plant configuration. The concept is essentially based on decentralized MES facilities realized by mobile and intelligent software agents that obtain their necessary information from an underlying plug-and-participate system. PABADIS covers the entire automation pyramid as it will be shown in chapter 5 - plug-and-participate technologies are used in conjunction with agent technologies in order to provide a solution that integrates all levels of a plant starting at the ERP level down to the field level. However, aside from the provision of a revolutionary automation concept for next generation automation devices, an equally important aspect is tackled by PABADIS: The solution is open for almost all plug-and-participate technologies as well as for agent technologies and must be compatible with legacy automation devices.

During the PABADIS research, specifically the part done at the University of Marburg, industrial requirements and prerequisites were determined and evaluated with respect to flexible plant networks created in ad-hoc fashion. This evaluation phase likewise verified the assumption regarding the huge number of different automation platforms (see also [4]). In order to cope with this variety of different platforms and with dedicated problems regarding stable, robust and easy-to-use software, the PABADIS concept follows the paradigm of platform independency: A secondary intention of PABADIS is to introduce Java into the field of plant automation (see figure 9 in chapter 5.2) because of its platform independency (see furthermore [1]). This led to a concentration of the PABADIS evaluation process on Java-based technologies and Java platforms supported by the automation devices. The major outcome of this evaluation is that the (Java) platforms provided by legacy automation devices are often limited device (Java) platforms, and furthermore such devices, on which the platforms run, are often characterized as limited devices.

Remark: The term 'limited device Java platform' refers to those Java platforms which are suitable for environments where full Java Virtual Machines (JVM) as well Java class libraries cannot be hosted and used. E.g., devices, on which only limited device Java platforms run may not provide enough memory and computational resources, appropriate display size, etc., required to run fully equipped JVMs.

The consequence of this limited-device character of target platforms and their limited device Java platforms is the development of a limited-device suitable concept. For the University of Marburg this led to the development of a limited-device suitable, Java-based plug-and-participate technology and a specific plug-and-participate concept for PABADIS in general. Both, the automation related plug-and-participate concept as well as the limited device compliant plug-and-participate technology are the main contribution of the author to PABADIS: The automation related plug-and-participate concept mainly comprises an abstraction layer that allows the horizontal integration in the automation field, namely the provision of a homogeneous view to the automation functions. Moreover, this innovative plug-and-participate application allows rather stable and robust infrastructures of machine functions that are abstracted from their particular hardware. The resulting infrastructure is finally the major basis for PABADIS' innovative planning facilities.

The second outcome of this research is the so-called Pini technology that runs on the "Kilo Virtual Machine with the Connected Limited Device Configuration" platform (KVM/CLDC). This technology finally relies on a general plug-and-participate interface abstracting the basic facilities of this technology genre. These results as the main contribution of the author to the project will be explained in chapters 5, 6 and 7 in more detail.

Comparing the focus of this thesis and the PABADIS approach, it is obvious that the outcome of this work completely covers the plug-and-participate aspect of the project. In other words, this thesis documents the research efforts done by the author within the PABADIS project as well as several investigations of related fields. This led to three particular facets for this work, namely the field of plug-and-participate technologies, automation solutions and limited devices. The goal of this thesis is therefore the provision of a limited-device compliant plug-and-participate-based automation solution using a suitable plug-and-participate technology. Major aspects in this respect are an automation related plug-and-participate interface abstracting the automation requirements as well as a specific architecture of a plug-and-participate technology providing

- service description mechanisms for service provision,

- a distributed community management architecture and
- a pattern for an efficient service use.

The starting point of this thesis is the evaluation of recent middleware and plug-and-participate technologies, followed by a survey of existing automation concepts. The major part of this thesis given in chapers 5, 6, and 7 finally shows the PABADIS concept, its particular plug-and-participate solution, as well as further aspects of plug-and-participate on the field level. Chapter 6 will provide detailed information about the Pini technology, while chapter 7 documents the measurements and results, which shows Pini finally as a limited device compliant plug-and-participate technology for the automation field.

CHAPTER 2

Plug-and-Participate Technologies in General

This chapter gives an overview of plug-and-participate technologies, their general features and principle objectives. It furthermore provides details of currently-available technologies - this covers one facet of the thesis. The starting point is a description of the general evolution of plug-and-participate technologies comprising the first efforts in this field. Especially Plug-and-Play technologies will be briefly touched upon, and the evolution to distributed systems based on software plug-and-participate concepts will be shown. This survey demonstrates plug-and-participate technologies as a subset of middleware technologies providing a certain degree of transparency to users of distributed systems. Subsequently, common middleware technologies like CORBA, RMI, COM and DCOM, which can be considered as first efforts in the area of distributed systems, will be depicted. Such efforts finally led to software plug-and-participate systems which are evaluated in the last section of this chapter. In the main part of this chapter, a general and abstract plug-and-participate interface is designed, which is derived from the definition. Furthermore, two particular solutions will be presented. Both technologies as well as the abstract interface are the main fundaments of a plug-and-participate solution for the automation field as shown in an exemplary fashion in chapters 5, 6, and 7.

2.1 From Plug-and-Play to Plug-and-Participate

The starting point of further discussions of this thesis regarding plug-and-participate technologies is the description of their basic idea. It is followed by an overview of the most important differences between the original Plug-and-Play concept and technologies generally summarized by the term “plug-and-participate” technologies.

The idea of modular systems is rather old and their basic objective is to avoid disadvantages of monolithic systems. Both types of systems, monolithic and modular, can be seen as orthogonal to each other - advantages of modular systems are generally the disadvantages of monolithic systems and vice versa.

One of the most important benefits of modular systems is the interchangeability of modules. They can be reconfigured depending on specific demands. For example, modules can be added to a system in order to improve its functionality. Such additional modules need to be integrated into the system, and the functionality must be offered transparently to other components. This process can be summarized as (re-) configurability. However, such reconfigurability may affect the entire system and thus needs to be as efficient as possible. In the field of computer hardware, Plug-and-Play technologies, for example, are introduced, providing a certain degree of flexibility and comfort. Flexibility in this respect reflects additional hardware components like network adapters, graphic adapters, memory sticks, etc., which can simply be added to (plugged into) a system. The term "comfort" refers to the fact that a user does not have to care about the extended system configuration, specifically the installation of driver software necessary for using these new components. In contrast, the system itself should be able to detect new hardware components in order to install the required driver software automatically and without any (human) intervention (for technologies offering those features please see [18, 46, 47, 55]). These benefits result from non-static and extensible systems - prerequisite for this concept is an open system that allows the combination of hardware of different vendors rather than being bound to hardware of only a specific vendor.

Although no dedicated and standardized definition of Plug-and-Play exists, a relatively common definition of Plug-and-Play is given by ANS T1.523-2001, Telecom Glossary 2000 (see [2]):

> *plug and play: Of or pertaining to the ability of certain operating systems to automatically (a)* ***detect*** *a new device that has been added to the system, (b)* ***uniquely identify*** *that device, and (c)* ***install*** *the appropriate drivers and system files for that device. Note 1: Identification is facilitated by means of predetermined identification numbers hard-coded into the device. When the operating system boots up, it polls all installed devices and checks the returned identification numbers against the list of previously installed devices. If an identification number is not on that list, the number is looked up in a master database (either locally or on-line) and the new device is identified. Note 2: Not all devices are plug-and-play compatible.*

Mechanisms like these are not bound to hardware systems only. Likewise software systems are generally modular systems, at least since the object oriented programming paradigm (see also [91] for the

object oriented programming model) was introduced and became widely used. This idea of modular software systems today likewise shows a trend towards distributed systems. They usually require mechanisms that provide infrastructures in an efficient and simple way without unnecessary configuration efforts: System components should be as autonomous, and thus as independent as possible from other specific components. A possible way to provide the required mechanisms abstracts the idea of hardware Plug-and-Play systems and adapts them to the needs of distributed software systems in order to alleviate the handling of such systems. Although the goal of hardware Plug-and-Play is similar to the required software counterparts, the original hardware-related concepts need to be adapted to software needs: While Plug-and-Play systems mainly provide hardware-related configuration capabilities, software-related concepts usually aim at providing services to appropriate systems. These services are generally neither hardware-, nor device-, nor location-specific, and thus not location-bound, but instead are abstracted as arbitrary system components: Components using these services are more interested in a certain functionality than in a specific device. The second aspect of service-related distributed sofware systems is the provision of appropriate access patterns for service users (clients). This also reflects mechanisms for clients to find/search for required functionalities.

Such software systems, therefore, abstract these rather hardware-related Plug-and-Play features so that the corresponding software concepts generally allow the alleviated configuration of complete infrastructures in a spontaneous manner. The resulting infrastructures are built by means of their components that participate in the community/infrastructure.

Systems of this type usually consist of active components that organize service - service user relationships in a spontaneous fashion. This leads to completely different behavior patterns of infrastructure components. In conjunction with the enhanced functionality, also the term “Plug-and-Play” has been harmonized with the provided capabilities - those technologies are summarized by the term **plug-and-participate** technologies:

The term "participate" reflects the general behavior of components of those systems - plug-and-participate systems handle all components in the same fashion. For example, components are able to join and leave the system freely, to offer and use services actively and to form spontaneous infrastructures. The spontaneity of this infrastructure creation allows components to participate in completely unknown infrastructures. Relationships between the components are defined ad-hoc depending on the components already participating at a given moment.

Such system behavior requires a certain semantic in order to allow proper global functionality. The system semantic is defined first by the semantic of single components of the infrastructure, and second by established relationships. The configuration of such relationships can finally be ascribed to the local semantic of components because they actively trigger the establishment of relationships.

From this given description and the definition of hardware Plug-and-Play, the following definition of software plug-and-participate systems is derived by the author:

> *Plug-and-participate describes the ability of modular systems to form spontaneous networks; specifically the creation of ad-hoc service infrastructures is one of the main issues covered by plug-and-participate technologies. This concept offers certain capabilities to the components:*
>
> - *ability to join the community autonomously and identify the components uniquely*
> - *ability to leave the community without any coordination effort, but in consideration of system robustness*

- *mechanisms to find other components*
- *mechanisms to use the components appropriately without the (expensive) installation of driver software*

The ad-hoc facet of the concept allows components to participate in unknown infrastructures, and furthermore to obtain necessary information at runtime. As a consequence of these capabilities, plug-and-participate technologies allow modular systems to:

- *establish relationships between components triggered by the components themselves*
- *define infrastructure semantics based upon the semantic of single components and relationships inside the community.*

This definition concludes this overview of the evolution from Plug-and-Play to plug-and-participate. However, plug-and-participate does not replace hardware Plug-and-Play; it instead applies the concept to distributed software systems in an abstracted and enhanced fashion - plug-and-participate has its roots in the necessity of modular systems to be reconfigurable. The first evolution step was the introduction of Plug-and-Play mechanisms in the field of (computer) hardware. Its objective is the provision of alleviated system configuration mechanisms by automatically determing extensions of systems, and installing respective driver software without manual intervention. This hardware-related concept is adapted to software systems as a second evolution step with respect to the trend towards distributed computing and distributed systems. Resulting plug-and-participate technologies abstract the specific hardware-related view to a more functional view of services and service users. Both are generally neither location- nor device- nor hardware-bound. Such a concept covers the entire field of modular systems beginning at the service provision up to the establishment of relationships between services and their service users. One of the most important aspects in this is that components do not need any information about the community they want to participate.

2.2 Middleware Technologies

The intent of this section is to show the basic idea of middleware technologies, and thus to give an introduction to that theme followed by a description of legacy technologies that are still in use. These legacy technologies are the basis of modern plug-and-participate solutions, but also of the automation solutions shown in chapter 3. Moreover, the technologies focused in this section belong to the different epochs of the middleware field - their history is reflected in their sequence in this section: COM/DCOM are the oldest concepts in the middleware field. CORBA also belongs to relatively old technologies, but due to its ambiguous, generic and powerful features, it is still considered state-of-the-art. RMI and SOAP are rather modern, applying recent object-oriented facilities.

As outlined in section 2.1, plug-and-participate systems are the evolution step after pure hardware Plug-and-Play or, more abstractly, plug-and-participate is an application of the abstracted and enhanced hardware Plug-and-Play concept to distributed software systems. The motivation for both technologies can be found in the increasing complexity of systems as well as the increasing need for appropriate flexibility: In the same way as the computer hardware becomes more and more complex over the time, likewise the complexity of software also increased tremendously. Such an increased software complexity is driven somewhat by the increased capability of the hardware, but mainly by the

increased complexity of problems to be solved - monolithic software systems are too complex and expensive to develop, too inflexible and too expensive to maintain. For example, the adaptability to new problems appearing often and quick, the integration of new functionalities and therefore the maintenance became rather difficult and complex. Due to such shortcomings of monolithic systems, the evolution step in software development introduced the paradigm of modular, object-oriented programming. This concept also has an impact on network technologies, specifically network-based and distributed applications: Different modules running in different address spaces or even on different hosts provide the needed functionality. These modules must cooperate/coordinate their work so that the functionality can be provided appropriately. Concepts like client/server and recently service/service user became commonplace and revolutionized the way of thinking in software development. The concepts, of course, are not bound to networked applications, but in the same way, they can be carried over to distributed software systems that generally rely on interactions and communication between their modules.

However, these concepts not only solve problems by providing a certain degree of flexibility, they also require new patterns in application design and use. For example, the service - service user concept requires a more abstract view of the entire system than a simple component-based view: A service user does not use a (hardware) component providing a certain service but instead uses a needed service independent of its location/implementation/component. The same arguments apply for client/server applications: A client requests a server for a specific task and is usually not interested in how this task is solved by the server or how the communication to the server works in detail.

In comparing both approaches - the service/service user and the client/server approach - an important difference becomes obvious: The latter mentioned pattern requires that the client has explicit knowledge of the server location, and thus an arbitrary component - component relationship is established. This connection is configured by the components but not by an underlying system. The service - service user concept, in contrast, defines a service user (component) - service (infrastructure) relationship, wherein a user receives a reference to the required functionality. Such a reference can be provided by a dedicated component but also by a respective infrastructure. However, the user simply holds a reference instead of detailed information about the location of the functionality and hence does not need to care about how to connect to the service.

This simple example can be abstracted to a common design pattern: the provision of transparent relationships between system components. Such a pattern usually covers several layers of interaction, e.g., the physical connection between components that cannot be removed but must be masked. The procedure of masking the direct connection is simply aggregated to an additional level, thus ensuring the required transparence. It must hide the particular access to specific components and allows therefore components to focus solely on their dedicated tasks like using a service. This layer on top of the physical connection layer provides the required transparency and is situated beneath the application layer. Such a middleware facility is defined in the following formal way:

> *Middleware refers to a broadly and loosely defined software genre whose programs sit between an application and an operating system* (see [33]).

Keeping the features and the definition of plug-and-participate systems in mind, the middleware character of those technologies becomes obvious: They focus on the creation of spontaneous service infrastructures and the transparent access to these services. Such access patterns, however, do not necessarily need to be further defined by the plug-and-participate technologies. An evaluation of the

requirements of this procedure recommends the use of arbitrary middleware concepts in order to keep the transparency - section 2.3 describes two examples of plug-and-participate technologies.

2.2.1 COM - The Component Object Model

One of the first concepts in distributed object systems is the Component Object Model (COM, see also [57]). It mainly focuses on the support of users in building distributed object applications having a certain transparence. This generally covers the creation of objects in environments, while the second and more important objective of COM is the support of users in arranging relationships between the objects. Such relationships basically describe interactions between objects in order to participate in a broader system, and thus to benefit from the functionality of the respective infrastructure. Both aspects, object creation and object relationship setup, abstractly show the major goals of COM. Following the basic objective of the middleware field, namely the provision of transparent systems, COM's facilities for object creation and object relationships are designed in a compliant fashion: Transparency in this respect mainly reflects the ability of objects to request other objects, particularly the functionality of objects of interest, but not their specific implementation. This feature is improved by COM's independency of any programming language. A further basis of COM also ensuring transparency is the use of interfaces to specify the functionality and the general behavior of COM objects. For example, a request to a COM system is defined by the interfaces of a demanded object. Responses of a COM system on such requests, in turn, provide the references to those objects meeting the request. Such a behavior pattern implies that COM objects must be registered with the system in order to be accessible. A returned reference is simply specified by its (COM) interfaces, and thus the required transparency can be ensured. Moreover, a reference to an object is the basis for the relationship between objects - they can access each other using the references. This access process, however, is not further defined by COM. It is usually implemented via a simple client - server connection.

In addition to these general facilities - the creation and registration of objects and the object connection setup - COM offers appropriate mechanisms for object management such as garbage collection and reference counting, and also security mechanisms are defined. This basically refers to the definition of standard COM interfaces and mechanisms that must be implemented by COM objects in order to participate in a COM infrastructure (see also [107]).

2.2.2 Distributed Component Object Model

While the COM architecture outlined in the previous section is defined for distributed object systems within a certain address space, the Distributed Component Object Model (DCOM) enhances this model to provide object distribution across different address and name spaces (DCOM is defined in [55]). The main focus of this enhancement comprises the major facilities of COM as well as extensions and adaptations of the standard COM interfaces. For example, the mechanism for object creation has been extended by the possibility to specify the server location, at which an object should be launched. Also object interaction facilities have been enhanced so that objects can invoke methods on other, remote objects. Of course, this enhancement essentially follows the transparency requirements of middleware technologies as already described for COM. The essential issue pertaining to the method invocation facility is the transmission of invocation parameters and return value objects between communication partners. In particular, marshalling mechanisms are defined that convert the object data to

interpretable byte streams. Those streams can easily be exchanged between interacting objects (see also [42]).

The distribution of objects across different name and address spaces furthermore requires adequate mechanisms regarding object management and security. An important issue is reference counting, and accordingly garbage collection mechanisms - specific COM standard interfaces tailored to local object management facilities are extended to accomodate distributed object management needs.

2.2.3 CORBA - Common Object Request Broker Architecture

In the same way as described for COM and DCOM, the fundamental objective of Common Object Request Broker Architecture (CORBA, specified in [52]) is the provision of transparent infrastructures for objects. However, despite the abstract similarity between CORBA and COM/DCOM, there is an essential difference: While COM/DCOM are basically fixed to particular object systems, CORBA is open for almost all object systems - it defines a concrete object model for its specific protocols and facilities, but not an implementation specification. This results in a declaration of a special behavior of CORBA components: Client and services/servers are distinguished in the same way as in COM/ DCOM. CORBA furthermore specifies a component called an Object Request Broker (ORB), which is the central instance of the CORBA concept. Both, clients and services can be from any object system, while the ORB supports these objects in their interactions regardless of their object system. This requires suitable (ORB-related) mechanisms in order to ensure CORBA's main philosophy: Interoperability is more important than portability (see also [14], section “CORBA as a Standard for Distributed Objects”).

In its most general form, the ORB acts as a broker between the objects - it manages request mapping, and thus guarantees the desired interoperability: If the ORB receives a request from a client for a dedicated object, the ORB routes this request to the addressed object. This process involves several specific mechanisms on the client side as well as on the service side in order to overcome possible differences in the object systems. For example, (service) objects must be registered with the ORB. This procedure is called “initializing the object with the ORB”. Client entities can request the ORB in order to obtain references to these registered objects. However, the main focus is on request broking: On the client side, the ORB receives the request for a service object through a stub mechanism. Stubs are either defined statically through the interfaces of the service object, or through a so-called Dynamic Invocation mechanism. The required interoperability among different object systems and the transparency necessity is tackled by defining these interfaces with the Interface Definition Language (IDL, see [61]). Using IDL for interface definition allows a complete independence of client and service environments, which can rely on different object systems. IDL and the ORB mechanism that connects both “worlds” finally ensure this independency. After an ORB has received a request, it routes this request to the addressed service object and dispatches the request using ORB internal mechanisms. These mechanisms can either be defined in the ORB core (mechanisms that are the same for all ORB implementations), or can be ORB-specific features. The service addressed in the interaction receives the request through the counterpart of stubs - the skeleton mechanism. Here likewise either static and service-specific skeletons are defined by their IDL service interfaces, or a dynamic skeleton can be applied. Which mechanism is finally used depends on the ORB. The service finally evaluates the request, but this is out of the scope of CORBA. This procedure of request routing must deal with possible differences in object systems - an additional layer called an Object Adapter, which masks the differences between different ORB cores, is used for that purpose. These differences can result from using

different object systems for ORB and services, and thus must be masked in order to ensure the required interoperability and transparency.

The interoperability, transparency as well as the field of application of CORBA is furthermore improved because of its independency of any programming language. However, a certain language mapping facility is required. Such a mapping is provided by IDL interface definitions and IDL compilers as well as by the Object Adapter instance.

In addition to these more "single system" bound interoperability issues, ORBs also must be interoperable between each other. This is the basic objective of the General Inter-ORB Protocol (GIOP) - it is a rather abstract protocol and does not specify any detailed communication technology. An implementation of GIOP is provided by the Internet Inter-ORB Protocol (IIOP) that provides a TCP/IP based GIOP realization (for IIOP and GIOP refer the CORBA specification [62], specifically chapter 12, and [60] for IIOP especially).

2.2.4 RMI - Remote Method Invocation

The previously described technologies mainly focus on object creation and object interactions, but they are not fixed to a particular programming language. In contrast, the Java Remote Method Invocation (RMI) concept is tailored to the Java programming language (see [98]). It offers facilities for method invocation between (Java) objects situated in different Java Virtual Machines. Aside from the transparency requirement, the Java RMI concept mainly concentrates on adapting the local Java object model to a distributed model, so that it keeps the Java method invocation syntax and semantic. If an object is to be published to a distributed system, it must first be registered with the Java RMI naming service facility. Objects, that want to access those registered objects simply request the naming service in order to retrieve a reference and finally perform the desired interaction. This naming service access is implemented as a simple method invocation, which keeps the entire concept as simple, efficient and as transparent as possible (examples are given in [93]). Ensuring this simplicity of the concept, and keeping the Java syntax and semantic, requires that most of the processes involved into the interactions are hidden from the users. Among processes like physical connection set up, object localization and request dispatching, the main concern is on a Java syntax compliant transmission of invocation parameters and return values. Parameters are usually considered arbitrary Java objects, while Java RMI distinguishes between local and remote objects: Remote objects are allowed to be accessed remotely utilizing a stub of an object. A Java RMI stub represents its referred object in the system and is passed to the target system. Specifically, if a remote object must be transferred to another JVM, it is automatically replaced by its associated stub. In contrast, a local object is only valid in a local scope, and is thus transferred as a complete copy to a client. Independent of whether remote or local objects are to be transferred through the network, the crucial point lies in the transmission of a potentially complex object. This must be transparent for the users, and thus it must not differ from patterns for local method invocation. If the Java RMI and the Java environment receive an object to be sent to another host, this object is serialized on the sender side. On the receiver side, the object must be de-serialized and restored. Such restoring processes require all class codes appendant to the object, and therefore the Java RMI concept also incorporates an extended class loading mechanism allowing dynamic remote class loading (for serialization please refer to [99]).

To summarize the Java RMI concept and compare the concept with COM, DCOM and CORBA, always a similar focus of all four concepts can be observed: the provision of transparent distributed

object systems. Despite this similarity, the Java RMI concept differs essentially from the other concepts, which are not fixed to any programming language and thus cannot benefit from such a language. These concepts, however, benefit from their openness for different languages and their broad field of application. This is in strong contrast to the Java RMI facility, which smoothly integrates the middleware concept into the Java programming language. The concept itself, on the one hand, enhances the local Java object model towards a distributed model, and on the other hand, it makes extensive use of Java features such as object creation, (local) security and class loading.

Moreover, there exists an approach for connecting Java RMI and CORBA via IIOP - an overview of this topic is given in [94].

2.2.5 SOAP - Simple Object Access Protocol

The fundamental objective of the middleware technologies shown in the previous sections was mainly concentrated on the facilities for object creation, object relationship setup, and object infrastructure management. The major goal of these technologies has been always transparent object interactions supported by the middleware technology utilized. For example, facilities like finding communication partners and gaining references are the main objectives of the surveyed technologies. Such features are provided by the frameworks, while the contents, formats, rules and interaction patterns for the object interactions are not influenced by the related specifications. This is in strong contrast to the Simple Object Access Protocol (SOAP), which mainly focuses on the definition of the message exchange patterns (see the SOAP specification [7], which stresses these fundamental features). In particular, SOAP defines dedicated rules for an alleviated message exchange between objects and also incorporates patterns for simple method invocation mechanisms. The general structure of SOAP is kept as simple as possible. It defines an `Envelope` as the main container of the framework, appropriate encoding rules for the messages and finally the aforementioned method invocation patterns, namely RPC rules. In order to maintain this simplicity, a generic message format is defined that is completely independent of any communication technology. The generic fashion of the messages is ensured by using XML and a rather simple message format consisting only of `<Envelope>`, `<Header>` and `<Body>` elements. The `<Envelope>` is a mandatory element, and can be seen as the container/root of each SOAP message. The `<Header>` element, in contrast, is optional and can be used to define additional information. Of course, each SOAP message has to have a `<Body>` element containing the information of the message.

Due to the independency of SOAP on any communication technology as well as the generic message format, its field of application is rather broad. For example, messages can be exchanged via any arbitrary TCP/IP connection, but usually SOAP messages are sent via HTTP. In this case a message is simply treated as the payload of HTTP communication.

Although SOAP is a rather simple protocol offering simple mechanisms for message exchange, as it was defined as the major goal, several important details have been neglected in this overview. These details can be found in the specification given in [7].

2.3 Plug-and-Participate Technologies

After the evolution of general middleware technologies has been sketched including abstracts of four well-known middleware concepts, the focus will be moved to plug-and-participate technologies. This genre of concepts covers one main facet of this thesis so that the following subsections show appropriate details of these concepts. In the introduction to middleware concepts and also in the overview of middleware evolution, general plug-and-participate concepts are classified as members of the middleware field. Plug-and-participate technologies furthermore rely on them as it will be depicted in this section. Likewise, the basic objective of plug-and-participate concepts is stressed, namely the creation of ad-hoc service - service user infrastructures. These general statements regarding plug-and-participate concepts will be deepened first by describing an abstract plug-and-participate interface that is derived from the stated definition of plug-and-participate concepts. Particular implementations of this abstract interface are given by two specific technologies - each of them constitutes one extreme in the plug-and-participate field, because they apply completely different approaches:

- a decentralized service architecture, but using a central component for infrastructure maintenance and management
- a totally decentralized service architecture without any central component

This opposition of a totally decentralized architecture on the one hand, and a decentralized architecture using a centralized management, on the other hand, implies that the required mechanisms for transparent and ad-hoc infrastructure setup will differ essentially.

The technologies shown here are Sun's Java-based **Jini** network technology and Microsoft's **Universal Plug and Play**. Both of them facilitate the additional, but no less important feature of platform independency.

A further aspect aside from the plain architecture description is an evaluation as to whether the technologies are suitable for the field of industrial automation, at least from the technological point of view. The main focus of this lies in the determination of requirements but also in giving a first impression of possible fields of use. A detailed evaluation of the usability from both points of view, namely the technological and conceptual perspective, is given in chapters 5, 6 and 7.

2.3.1 An Abstract Plug-and-Participate Interface

The description and evaluation of plug-and-participate technologies requires an abstract basis. Such a basis should also alleviate the comparison of technologies by defining meaningful comparison patterns - in this case, common facilities of plug-and-participate technologies will be determined that are aggregated into an abstract plug-and-participate interface - as a result of the author’s work in this field. The following fundamental characteristics can be derived from the plug-and-participate definition:

- Entities must be supported by plug-and-participate environments in joining a community. For plug-and-participate technologies, this means the provision of a protocol for service providers to offer their abilities to the community in a transparent way.

- Spontaneously created communities must be robust in terms of fluctuation of components - entities will leave a community either normally or due to serious problems such as network partitioning or crashes of entities. Plug-and-participate technologies must be able to detect such entities and must keep the community as robust and reliable as possible.
- The counterpart of service provision is the search for specific functionalities - plug-and-participate environments must support their participants in obtaining references to dedicated services. This basically necessitates the definition of an appropriate lookup protocol.
- Plug-and-participate systems generally aim at providing transparent relationships between the entities that provide the services and the entities intended to use them. Such relations generally rely on using the offered functionality. This utilization is generally supported and alleviated by plug-and-participate technologies. For example, the references to service facilities can provide specific access mechanisms, or the environment can define a dedicated protocol for this purpose.

Additionally to these facilities derivable from the definition, plug-and-participate systems usually provide a certain information flow facility in their concepts. Such an information flow mainly embodies a mechanism allowing participants of communities to track the infrastructure and its members - any interested party is informed about changes in the infrastructure.

A sixth facility of plug-and-participate technologies concerns a reasonable description mechanism of service features - plug-and-participate technologies generally define a dedicated attribute mechanism allowing the description of services. Such attributes are essential for the service search facility.

These six facilities can be aggregated to an abstract plug-and-participate interface that can be implemented by all technologies. It furthermore provides a basis for comparison by specifying the abstract behavior but hiding the particular realization. Based upon this expected behavior, the specific implementations can be compared in order to determine their advantages and disadvantages. Weighting and measuring these “values” allow rather objective evaluations of concrete implementations.

2.3.2 Jini and the Jini Surrogate Architecture

One classical plug-and-participate concept is the Jini networking technology (see the specification in [3]), which was developed by Sun Microsystems in 1998 (the official introduction of Jini was on 25 January 1999). The intention/idea of Jini is to adopt the goal of Oak (see the specification in [36]), the antecessor to Java. Oak was developed as a programming language to tackle the problem of different platforms of common goods, especially refrigrators, set top boxes, digital TV, etc. These devices, which can be characterized as limited devices, should be improved by facilities such as exchanging and installing programs and connecting to appropriate networks (like home automation networks as shown in [48]). From such features, the products should gain added value through enhanced facilities. However, Oak failed. Accompanied with ongoing Internet technologies, improved common goods, etc., the former idea of a platform independent programming language has been resumed - the Java programming language was developed as a common Internet programming language. With ongoing Internet development, and efforts in the area of common networking facilities of arbitrary devices, likewise the former intention of enabling devices for networking has been resumed: The Jini technology was introduced, which shows nearly the same target area as Oak - the entire field of common goods was defined as the original target area of Jini. Jini moreover not only resumes the basic idea/concept of Oak but enhances it with respect to the evolution of the requirements and the technology. This enhancement led to a generic and abstract technological concept providing the spontaneous network-

ing of services. Due to the inherent resource requirements of the ambigous and powerful concept, the target area of common goods (~limited devices), however, is not met. As a consequence of this observation, the Jini concept has been extended by the Jini Surrogate Architecture, which is shown at the bottom of this section.

In order to become familiarized with the Jini technology, first an overview is given including a description of the basic architecture and mechanisms. This overview is followed by examples for simple clients and services illustrating the behavior of arbitrary Jini systems. These examples likewise show the use patterns of common Jini protocols that in turn are the contents of section 2.3.1.4. Subsequent to this protocol description, the Jini Surrogate Architecture will be sketched. A final requirement analyzis is aggregated with the UPnP technology in section 2.3.4 in order to provide a comparison of both technologies.

2.3.2.1 Overview of Jini

Jini is a (classical) plug-and-participate technology originally designed for making devices - especially limited devices - network-compatible and more "intelligent". This ambiguous goal can only be reached, if the wide variety of different platforms being the state of the art in this field is tackled. A solution that overcomes this problem must be platform independent, and for this reason, a Java-based plug-and-participate technology was developed: Jini provides mechanisms to build communities of services rather than simply providing mechanisms for (limited) devices to join networks. This implies an abstract view to infrastructures, and ensures an important advantage with respect to the required transparency within those systems - users are normally interested in a certain functionality (transparently provided by a device) rather than in specific devices themselves.

These arguments describe Jini as a ***Java-based plug-and-participate technology*** providing ***ad-hoc networking facilities for service infrastructures***: Services join the network using respective Jini protocols, while clients use Jini facilities first to gain an overview of the community and, second to gain access to required services. Due to Jini's ad-hoc feature, neither clients nor services need any knowledge in advance about the infrastructure they wish to join. This behavior is not a specific Jini feature because plug-and-participate technologies naturally have to provide such facilities (as manifested in the definition of plug-and-participate). These natural plug-and-participate features are implemented as Jini-specific protocols that are used to establish Jini infrastructures consisting of services, service users and a central instance. This central instance is responsible for infrastructure management. For example, it allows Jini entities such as Jini users to obtain an overview of the Jini community and to retrieve references to dedicated services. Such a lookup functionality furthermore mandates that services register with the central instance in order to be available in the commmunity. A direct consequence of this general behavior of the Jini entities is that they always require at least one reference to a central Jini instance. This instance is smoothly integrated into the architecture, since it behaves like any arbitrary Jini service. Corresponding to its main facilities, namely the provision of an overview of the community, the instance is called the Lookup Service (LUS).

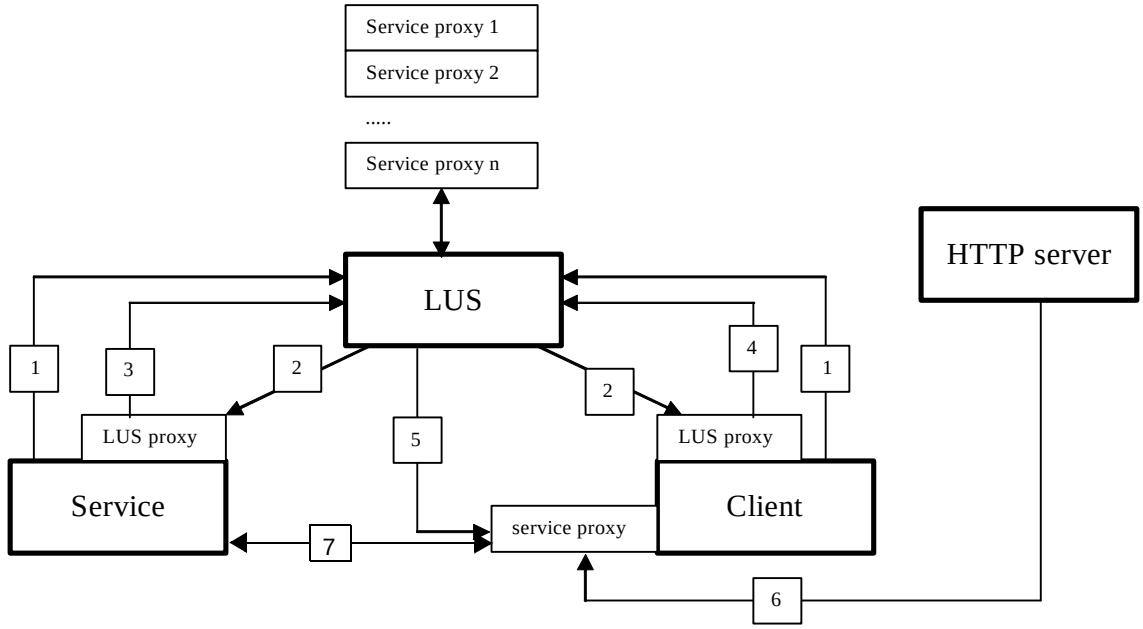
The ad-hoc facility of Jini allows clients as well as services to participate in a Jini community without advance knowledge of LUSs. Entities retrieve the necessary information during runtime using the Jini discovery protocol. The result of a discovery is a reference of one or more LUSs (if available) that is the basis for participation in the Jini infrastructure: For example, services need such reference to join the community, while clients use it to obtain an overview of available services or even to obtain a refer-

ence to a dedicated service. A community join process is simply described by passing service-related information to the LUS such as a `ServiceID` (if already available), the service proxy object and an attribute set. Clients, in contrast, request the LUSs for services by performing the lookup protocol. It usually finalizes in delivering a reference to a service, namely the `ServiceID`, the attribute set and most importantly the service proxy object. Such a service proxy provides the clients with the access path to the service - the proxy itself acts as a broker between the service and the clients. Behavior patterns like this require that the proxy object implements the service interfaces in order to ensure a proper service provision. This is furthermore the main basis for the Jini lookup protocol: Clients usually specify services of interest via the interfaces as the recommended alternative. Further alternatives for specifying services of interest are in passing either a `ServiceID`, an attribute set or even wildcard values to the LUS. Wildcards, defined by the `null`-reference, match all possible values. If, for instance, all search parameters are set to `null`, a lookup request matches all registered services (more details on the lookup are in section 2.3.2.4).

These basic plug-and-participate facilities sketched in the previous paragraphs directly refer to the interface defined in the section 2.3.1. In Jini, three particular protocols offer these facilities, namely the discovery protocol supporting entities in finding LUSs, the join protocol used to publish services to the community and the lookup protocol to retrieve service references. The main prerequisite for service use is the service proxy object - it transparently connects clients and services. Such an object must be transferred to the client. This proxy transmission is essentially based on the Java object serialization mechanism (see [99]) - a proxy is transferred as a stream of bytes extracted from the proxy object. It must be restored on the receiver side so that the proxy object can be used as service access path. This process requires the class code, which must be loaded either locally or remotely from the service's code base (usually via HTTP). Both, the recovery process as well as the class loading, is fundamentally based upon the Java RMI facility because the proxy is considered a local Java object that is transferred as a copy. If the proxy requires a connection back to the service, it can utilize a middleware concept like Java RMI. In this case, the proxy accesses the corresponding service stub of the service; the service is handled as a Java RMI remote object. Other, non-middleware communication patterns are also allowed because the Jini specification does not define the connection between the proxy and the service object.

The entire process, starting at the serialization of the proxy object and replacing the distributed remote service object by its stub, the subsequent de-serialization including the remote class loading process and finally the provision of the service facility to a client, is transparent for the infrastructure. Entities do not need to care about any of these steps because they are transparently provided by the Java environment, Java RMI or even the Jini framework itself. This transparency can also be derived from the

following figure, which describes the general Jini architecture but hides the accompanied infrastructure processes:



Legend:

1	issuing discovery requests
2	downloading the LUS proxy
3	joining the community and uploading the service proxy to the LUS
4	requesting the LUS for a service (lookup)
5	response of the LUS, including download of the service proxy object that in turn has to be recovered.
6	downloading the class code from the specified code base
7	service use via the service proxy and a connection back to the service

FIGURE 1. The basic behavior of a Jini environment

Although this basic architecture comprises the most important features of Jini plug-and-participate in terms of transparent service infrastructures, it requires facilities ensuring the robustness and stability of such architectures, as well as support a certain information flow facility regarding the status of an infrastructure and its involved entities. In Jini, these fundamental aspects of distributed object systems/ service infrastructures are provided by two dedicated protocols:

- leasing
- remote events

One of the most important advantages of distributed systems is the decentralized provision of functionality that for instance allows the load of the entire system to be spread among several nodes. The benefits of distributed systems, of course, not only mean load balancing among the nodes but also a distributed task fulfillment: A task can be split into subtasks that are processed at several nodes using the required functionality. Interactions of tasks usually involve the entire infrastructure, which is organized as a set of entities rather than as a single entity offering all necessary functionalities. The essential consequence of this distribution is that also failure sources are distributed across the system: Failure sources of a single node are basically limited to the functionality provided by this local node. If such a node fails, normally the global system is not affected by such a local failure, but only a small part. This means that a failure in one component does not necessarily cause a malfunction of the entire system, as it is more likely in centralized systems. However, local failures might harm the system if they stay undetected and especially if they diffuse throughout the system. Similar to centralized systems, in distributed environments failures can occur in every component, but these (distributed) components are more or less independent of each other. The consequences of occurring failures are completely different: While in centralized systems, failures often affect the entire system, in distributed architectures the so-called partial failures occur that usually influence only a specific part of a system and therefore they are rather difficult to detect. Moreover, in distributed systems failures can occur that may even be impossible in centralized systems, such as the disappearing of components, communication latency, network partitioning, etc.

A definition of this genre of failures is given below:

> *A partial failure occurs if one or more component of a distributed system fail(s) but other components cannot react appropriately or are not informed about the failure. Thus, the system as a whole appears to be working properly. Often, such partial failures cannot be detected immediately due to the system's distributed characteristics. Consequently, a partial failure only affects system behavior when other components try to access the failed components* (see [30], section 2.1.2).

Failures of this type need to be handled or at least detected in order to keep the system reasonably stable. A common way to solve this partial failure problem in distributed systems is the periodic re-allocation of resources or, more abstractly, the periodic announcement of the availability of a certain entity. In Jini, this facility is implemented by the so-called leasing protocol, which defines all resources, namely registrations with the LUS within a time constraint: Entities have to re-allocate/re-register periodically in order to continue participation in the community. Otherwise, if a lease duration expires, the lease is deleted and the resource is released. If the resource was occupied by a service, the service is removed from the community so that it is no longer available for clients. If the resource has been reserved by a client, e.g., for remote events (shown in the following paragraph), the client is removed from the passive information system.

Accompanied with the distribution of functionality and the possibility of partial failures, also special information flow facilities within distributed systems need to be handled. Due to the independency of components because of the distributed nature of such communities, the appearance/disappearance of components is transparent for these participants. Such a transparent behavior of distributed communities requires a mechanism that allows entities to be always up to date - they receive events if, e.g.,

new services join the system or services simply disappear. The Jini mechanism is called remote event notification. It is essentially based upon the Java event model, but enhances this (local) model for distributed event handling: An entity has to install an appropriate event listener that is responsible for receiving the events. Detecting the events, however, requires a further instance, which checks the status of a community and which decides to deliver events. This instance can also be a distributed system or a central instance. The Jini architecture provides this facility via its LUSs - they always have an up-to-date view of the community. Interested Jini entities, therefore, have to register with the LUSs in order to participate in the remote event notification system. Of course, the registration allocates a resource, and thus is also associated with a lease in the same way as described for service registrations. During the registration process, the entities provide nearly the same information as described for the lookup process in order to specify events of interest. If a LUS detects a variation in the community, it checks whether matching registrations exist. It finally dispatches the events to interested parties by connecting to their event listener instance - Jini defines the use of Java RMI for this purpose. The result of the remote event notification is usually the information about a new, a disappeared or even modified service - a Jini remote event object basically consists of the same information as described for the lookup return values, namely a reference to the concerned service, or the `null` reference, if the service disappeared. In comparing both protocols, the lookup protocol and the remote event protocol, a basic difference can be noted: The lookup protocol specifies an active facility for retrieving service references and an overview of the community, while the remote event mechanism is a passive one - entities passively receive information.

These protocols and architectural relationships of Jini are the basis for the following sections, which will introduce examples of a simple Jini service and client. Both examples are intended to reader's deeper understanding of the the mentioned protocols and to demonstrate the relationships within the Jini architecture. They furthermore illustrate how to use the Jini protocols, and how to implement Jini applications. Based upon this knowledge of the relations among the protocols and their particular use patterns, the subsequent section 2.3.2.4 will finally give an under-the-hood view of the different Jini protocols.

2.3.2.2 A Simple Jini Service

The simple Jini service example that is shown in this section demonstrates the use of the most fundamental Jini protocols such as discovery, join and leasing. The service itself offers a routing functionality for work pieces that request the service for routes between different processing nodes. Its service proxy basically acts as a broker, and thus forwards the client requests to the service. This communication is realized via RMI, which keeps the service proxy rather simple:

```
public class RoutingServiceProxy implements IRoutingService{
  IRoutingService_s service;

  public Route getRoute(Point start, Point dest, Criteria[] crit){
     ...
     route = service.getRoute(start,dest,crit);
     ...
     return route;
  }
}
```

CODE EXAMPLE 1. The service proxy of the example service

The previous code example also shows the basic structure of the `IRoutingService` interface, whose `getRoute(..)` method is the essential method. This interface, in turn, must be at least implemented by the service proxy providing the service functionality to clients.

In addition to these service related facilities, respective Jini interactions and facilities are more of interest in this example. They are defined in a framework embedding the particular service functionality. The most fundamental facility is the discovery part that in turn will also be used in the client example given in the subsequent section.

1. Performing the discovery protocol

Both, service and client use the most general discovery protocol requiring as little information as possible. It is implemented in the Jini standard class `LookupDiscovery`. The initialization of the discovery process is simply the creation of a `LookupDiscovery` object. Information regarding discovered LUSs are dispatched via a `DiscoveryListener` instance that must be added to the `LookupDiscovery` object:

```
LookupDiscovery lookupDisc = new LookupDiscovery(new String []{“”});
lookupDisc.addDiscoveryListener(new DiscoveryListenerImpl());
```

CODE EXAMPLE 2. Initialization of the `LookupDiscovery` class

If a LUS is discovered, the listener instance is informed by applying the Java event model, because a `DiscoveryEvent` is delivered. Such an event essentially consists of the LUS proxy that is the basis for further interactions such as the join protocol, lookup or remote event event registration.

2. Performing the join protocol

The join process generally refers to the provision of information describing the service within the community. In particular, the `ServiceID` of that service, its proxy object and an attribute set must be

passed to the LUS. These data are encapsulated in a so-called `ServiceItem` object that acts as a container for all service-related information:

```
ServiceItem item = new ServiceItem(null, new RoutingServiceProxy(this),
null);
```

CODE EXAMPLE 3. Creation of the `ServiceItem` object

The `ServiceItem` creation, shown in the previous code example, also demonstrates the instantiation of the service proxy object - it expects a reference to the service itself that represents a Java RMI remote object. With the help of this reference, the proxy will connect transparently to the service in order to provide the functionality. Once the `ServiceItem` object is created, it is passed to the LUS by invoking the LUS proxy's `register(..)` method:

```
ServiceRegistration reg = registrar.register(item,leaseDur);
```

CODE EXAMPLE 4. The registration process

Due to the service instance's remote object character, it is automatically replaced by its stub upon transmission to the LUS. The LUS response, namely the `ServiceRegistration` object, consists of the lease object that in turn is forwarded to the lease handling mechanism of this simple service. This process is shown in the next paragraph.

3. Performing the lease protocol

Each lease has to be renewed periodically parallel to all other actions of an entity; within this simple service example the renewal facility is implemented as a `Thread`, particularly in its `run()` method:

```
public void run(){
   while(!stopped){
     try{
(1)    long duration = lease.getExpiration();//when the lease will expire
(2)    sleep(duration - 10*1000);
(3)    lease.renew(60*1000);//renew for 1 minute
     }catch(Exception e){...}
   }
}
```

CODE EXAMPLE 5. The lease handling of this service

After the `Thread` is started, the lease duration is requested (line 1). Then, the `Thread` waits for the granted lease duration (line 2). When it awakes, it renews the lease (line 3). This renewal must specify the desired lease duration - in this case, a new duration of one minute (= 60,000 msec) is requested and is passed to the `renew(..)` method as a parameter. The entire mechanism runs in a simple loop that is continued until the service is completely stopped.

4. Providing the service functionality

The task of this service is to calculate a route from a starting point to a destination point within the plant. This means, the service needs an overview of all transport nodes with their connections between each other and their connected end points that are assumed as manufacturing machines. The service creates a graph of the transport system and calculates the respective path using the Dijkstra algorithm. The required knowledge about all transport nodes can be retrieved from an underlying plug-and-participate system like Jini (see the chapter 5.3.4).

2.3.2.3 A Simple Jini Client

The counterpart of the service described in the previous section is a client, which is topic of this section. While the basic behavior of a client is naturally the use of a provided service, the focus of this section mainly relies on the Jini related framework. This covers the discovery process, lease mechanism, lookup and remote event notification facility. Since discovery and lease handling were already sketched in the service-related section, this client-related section concentrates on the lookup protocol and the registration for remote events. Finally, the service use is demonstrated.

1. Performing the lookup protocol

The prerequisite of the lookup protocol is that the client holds at least one reference to a LUS. With the help of this LUS proxy, it is able to perform the lookup protocol: A `ServiceTemplate` object specifying the service(s) of interest has to be created. It expects a `ServiceID`, service interfaces that a requested service should implement and an attribute set, or the wildcard value (`null` reference) for any or all parameters. With respect to the service interfaces to be specified in the `ServiceTemplate` object, two important aspects must be considered:

- A client must know about the service interface of the demanded service; otherwise, it cannot access the service.
- The Jini service matching is fundamentally tied to Java types, and thus, the interface of the demanded service must be exactly that of the service.

In this example, the client specifies the `IRoutingService` interface but passes wildcards for the `ServiceID` and attribute sets:

```
ServiceTemplate templ = new ServiceTemplate(null,
                            new class[]{IRoutingService.class}, null);
```

CODE EXAMPLE 6. Creation of the `ServiceTemplate` object

This object is passed to the LUS via its proxy by invoking one of the following methods:

```
(1) Object o = registrar.lookup(templ);
(2) ServiceMatches matches = registrar.lookup(templ,count);
```

CODE EXAMPLE 7. The different `lookup(..)` methods of a Jini LUS proxy

In case (1), the requested service proxy is delivered directly to the client, while (2) returns a set of services (at most `count` services). These services are encoded in a `ServiceMatches` object consisting of an array of `ServiceItem` objects (carrying the `ServiceIDs`, the proxies and the attribute sets) of services meeting the request. From these services, the client can then choose the one it wants to eventually use.

2. Registration for remote event notification

The remote event facility of Jini is the passive way for clients to obtain references of services and to receive information about changes in the service community/infrastructure. In the same way as explained for the Jini lookup protocol, the basic prerequisites are the availability of at least one LUS reference and a `ServiceTemplate` object specifying the events of interest. A `ServiceTemplate`, however, only defines the features of those services that are of interest for the requestor, but not the particular event type. Therefore, a parameter must be provided that describes whether a service joins or leaves the community, or if a service simply modifies its attributes. The last parameter in this respect is the reference to the object, which will receive the events - a `RemoteEventListener` instance must be passed to the LUS. LUSs, in turn, observe the community, and if events occur they deliver them to the registered clients. This event dispatching in Jini is essentially related to Java RMI. Event listener instances are therefore Java RMI remote objects, and only their stubs are sent to the LUS, if the `notify(..)` method of the LUS proxy is invoked (see for more details the Jini specification in [3] and the RMI documents in [93, 98]).

```
EventRegistration reg = registrar.notify(templ, transitions,
                        new RemoteEventListenerImpl(),null,leaseDur);
```

CODE EXAMPLE 8. The registration process for remote events

As a response to a sucessful registration process, an `EventRegistration` object is received containing the assigned lease of this registration. This lease object is passed to the lease handling component as described for the service.

3. Using the service functionality

One main objective of plug-and-participate technologies is the transparent setup of service - service user infrastructures. The fundamental aspect lies in the transparency, and thus in an alleviated access to services. With respect to Jini, such service access is realized via the service proxy mechanism that either may provide the entire service functionality, or it can connect back to the service. In the latter case, it acts as a broker between the clients and a particular service. The proxy of this example relies on this broker facility and provides the service functionality by forwarding requests to the service: Assuming "o" is the service proxy object retrieved via the lookup protocol (as shown in code example 7), the service use is dedicated to a simple method invocation:

```
((IRoutingService)o).getRoute(point_start, point_dest, null);
```

CODE EXAMPLE 9. The service use, simplified to a method call

This example, of course, is only a very simple application of the Jini concept. However, the simple service as well as the corresponding client demonstrate the use of the most general Jini facilities, which normally are completely independent of the complexity of a service. These features, realized in dedicated Jini protocols, must be detailed in order to gain deeper knowledge of the Jini plug-and-participate concept, as well as to recognize appropriate problems that might occur if Jini is used in the field of plant automation.

2.3.2.4 Jini Protocols Under the Hood

After the principle use patterns of Jini protocols as well as an overview of Jini were sketched, the protocols must be investigated in more detail. This evaluation mainly covers the Jini implementation of the abstract plug-and-participate interface as defined in section 2.3.1.

Discovery

The Jini discovery protocol suite is the basic prerequisite for spontaneous networking - Jini discovery aims at discovering LUSs. This protocol suite and its protocols can be characterized by the behavior of the discoverers, their required information as well as the communication mechanisms used:

The **unicast discovery protocol** is an active protocol that requires information regarding an existing LUS. A “discoverer” sends a certain discovery message via unicast communication to this LUS, which listens on port 4160 for discovery messages. The LUS, in turn, responds with its LUS proxy object. Jini provides this facility in its `LookupLocator` class - its constructor and the `getRegistrar()` method are the essential features. A second representative of active protocols is the **multicast discovery protocol**, which in contrast does not rely on in-advance knowledge about LUSs. Discovering entities simply initialize the `LookupDiscovery` class of the Jini framework and install a `DiscoveryListener`. This `LookupDiscovery` instance automatically sends multicast discovery requests to the entire network - seven attempts every five seconds are sent to the Jini discovery multicast group (address 224.0.1.85, port 4160). If a LUS receives such a request, it usually responds with its LUS proxy - via unicast communication. If the `LookupDiscovery` mechanism receives this proxy, it generates a `DiscoveryEvent` object and delivers it to the installed `DiscoveryListener` instance(s).

After the `LookupDiscovery` class instance has sent the mentioned seven requests, it automatically switches to passive mode in order to participate in the Jini **multicast announcement protocol**. This protocol also belongs to those which do not rely on in-advance knowledge about LUSs. In contrast to the multicast and the unicast discovery protocol, however, the multicast announcement protocol shows a passive behavior. It relies on the fact that each LUS periodically sends announcement messages to the Jini announcement multicast group (address 224.0.1.84, port 4160). Each `LookupDiscovery` instance listens on this port for incoming announcements. If one is received, the listener component checks whether the announcing LUS is already known (and/or the list of known LUS is updated). In the case, if the LUS is not knwon, the `LookupDiscovery` instance performs the unicast discovery protocol in order to obtain the LUS proxy, which then is passed to all registered `DiscoveryListener` instances.

Remark: The LUS proxy is not contained in the announcement but simply the URL of the LUS. This saves considerable network resources.

Community Join

With the help of the join protocol, service entities enter the Jini community and make themselves available in the system. This requires specific information about the services like

- `ServiceID` (if one is already assigned to the service)
- service proxy
- attribute set

These parameters are encapsulated in a `ServiceItem` object that in turn is provided to the `register(..)` method of the LUS proxy. The proxy evaluates the `ServiceItem` object and converts the service proxy object into a Java `MarshalledObject`. Finally, the `ServiceItem` object is passed to the LUS as a copy of the entire object via an RMI method call of the `register(..)` method of the LUS service reference.

Remark: The Jini LUS represents a Java RMI remote object, and its proxy is a broker of requests.

The response on a registration request is a `ServiceRegistration` object consisting of the lease object associated to that registration (see also section 2.3.2.2 and [3, 30]) as well as further information related to this registration.

Lookup

The lookup protocol of Jini implements the service search facility as defined in the abstract plug-and-participate interface. It allows entities to obtain an overview of the community in an active fashion and finally the retrieval of references to services. Client entities can specify what services are of interest by either providing a `ServiceID`, if the required service is already known, or an interface set specifying the required service (Java) types, or they can provide attribute sets that a qualified service must match, or they can also define wildcard values. The parameters are encapsulated into `ServiceTemplate` objects that are, in the end, passed to the LUS via an RMI method call - the client invokes the respective `lookup(..)` method of the LUS proxy (see code example 7) that in turn forwards the search pattern to the LUS. At the LUS, the request is evaluated and finally the found services are responded to the requesting client. In particular, it returns the service proxy objects as Java `MarshalledObject`. There, the `MarshalledObject` must be restored in order to obtain the proxy object. This requires the download of the accompanied class files, as already stressed in figure 1.

In addition to this communication and request-related protocol part, the lookup protocol furthermore comprises a LUS-related part that is responsible for processing the request in order to check whether services are registered, which meet the requirements of the request. This matching-related facility will be described in more detail in subsequent paragraphs.

Robustness

The Jini lease protocol ensures a certain robustness within a Jini system. It is mainly concentrated on determining and removing inactive entities. For example, if an entity crashes or is disconnected from the network, it cannot leave the community appropriately, releasing all its occupied resources. Robustness reasons, as sketched in the abstract plug-and-participate interface, require suitable and efficient mechanisms that detect such malfunctions - in Jini, entities holding any resource have to send messages periodically in order to show that the resource will be occupied for a further period of time. Resources are associated with a certain period of validity. If such a period is exceeded, the resource is

released and the object's registration becomes invalid. This abstractly-described mechanism is implemented in the Jini lease protocol: Each registration is annotated with a lease object, whereby the lease duration is granted by the LUS. Using this lease object, entities re-allocate their resources if they renew the associated lease for a further period. If a lease expires, the LUS assumes that the respective owner is no longer alive, discards that lease object, and thus the former reserved resource is released.

Remote Event Notification

Plug-and-participate entities are generally independent of each other, and thus are unable to detect events and changes in their infrastructures. Ad-hoc networking technologies therefore provide mechanisms, which allow entities to be informed about events in such transparent systems. This required information flow is manifested in the abstract interface: Jini implements a passive information facility in its Remote Event Notication protocol. It relies on appropriate registrations of entities with LUSs for events of interest - they basically pass the same information to a LUS as specified for the lookup protocol, namely `ServiceTemplate` objects. Such objects only define the features of services affected by the subscription - entities furthermore have to pass information about the event type and a reference to a `RemoteEventListener` instance. This listener instance is, in the end, informed about occuring events. While the event type is simply an `int`-value describing whether a service joins or leaves the network or alters its attributes, the `RemoteEventListener` instance is generally a Java RMI remote object.

Remark: The Jini remote event notification protocol is strongly based on RMI. This reasonably shows a standardized system behavior and, most importantly, extends the local Java event model to a distributed event model.

The registration process is rather simple: the invocation of the `notify(..)` method of the LUS proxy. It returns an acknowledgement of the registration, namley an `EventRegistration` object providing access to this registration and the associated lease.

On the LUS side, the registration information is stored, especially the `RemoteEventListener` instance is resumed. Due to its remote object character, the reference to the listener is replaced by its stub. Nevertheless, the restoration requires the class file of the stub - remote class loading is required.

If the LUS detects a modification of the infrastructure, which fulfills the requirements specified in a registration for remote events, it invokes the `notify(..)` method of the stub and delivers a `RemoteEvent` object to the interested entity.

Jini Lookup Service

The LUS is the central Jini instance, offering a specific Jini service that provides managing facilities for the Jini community. For example:

- Services *have* to register with it, while clients *can* request information about the community from the LUS.
- It tracks the leases of the registrations.
- It delivers remote events to interested parties.
- It participates in the discovery protocol suite.

Such facilities, of course, require suitable data infrastructures that allow the efficiently storage of service registrations, service search, lease management and event recognition. For example, the LUS must store the `ServiceItem` objects in order to efficiently find services meeting a request or detect expired leases. Jini, however, does not specify these internal structures, and thus each LUS implementation might use different structures. The matching of services against requests, however, is not affected by the data structures: First, the LUS checks whether a `ServiceID` is specified, and if so, it searches for that particular service. If, in contrast, service types are provided, it compares the set of registered services with the requested interfaces. The provision of attributes finally causes the LUS to compare the stored attributes with those requested. This matching is based on the `MarshalledObject` representation of attributes, which leads to a (exact) byte comparison, and thus only exact matching is provided in this respect (more details are given in section 5.3.3.3.7).

The lease management facility of the LUS comprises two parts: The LUS must grant leases corresponding to the registration, and furthermore, it must track these leases. Granting leases is rather simple because it can be circumscribed to the definition of a duration. Tracking leases, in contrast, requires a `Thread` that awakes if either the shortest lease expires or a lease is renewed, and thus the shortest lease must be redetermined.

In performing these management facilities, the LUS implicitly satisfies remote event requirements: If a new service joins the community, or a lease expires or is cancelled, the LUS checks whether clients are interested in those events. If a service modifies its attributes, the LUS recognizes this also implicitly, because LUS internal data are affected.

A fundamental aspect of the management facilities is the participation in the Jini discovery protocol suite. This ensures the spontaneity of Jini infrastructure creation and maintenance: Entities participate easily in unknown systems, and thus they can benefit from those communities configured on-the-fly. Hence, a LUS must be:

- a member of the Jini multicast discovery group (224.0.1.85, port 4160) in order to listen for incoming discovery requests. In this protocol, the LUS takes on the passive role as described in the discovery paragraph - it simply responds with its proxy.
- a member of the Jini multicast announcement group (224.0.1.84, port 4160), where the LUS has the active role due to the periodically sent announcement messages.
- a LUS must be able to listen for incoming unicast discovery requests.

These three distinct roles are reasonably implemented by three `Threads`.

All these described LUS facilities lead to a specific sequence of the LUS initialization: First, it must initialize its data structures, so that the join, lookup, lease and remote event mechanisms can be utilized properly. The second step is the initialization of its discovery component, so that it can be discovered by arbitrary Jini entities. As the last step within the initialization phase, the LUS joins the infrastructure, because it is nothing more than a specific Jini service. It therefore provides a service proxy that implements the `ServiceRegistrar` interface, which is one of the fundamental properties of the Jini architecture.

2.3.2.5 The Jini Surrogate Architecture

One of the most important fundamentals of Jini is the ability of devices/platforms to execute and download Java byte code. Further prerequisites are the export of classes/objects to requesting parties and finally the ability to participate in the basic Jini protocols like discovery, join, lookup and leasing.

However, there is a category of devices (so-called limited devices) that are unable to satisfy these requirements, and therefore cannot directly participate in the Jini technology. For example, consumer devices often are not able to provide the required resources. Products such as set top boxes, digital TV, etc., cannot participate in Jini - Oak and later Jini were originally designed for this category of target devices. With the Jini Surrogate Architecture, such problems are addressed by extending the target area of Jini to the field of limited devices (see [97]). It defines the means for those devices/ components to participate in Jini. The central point of this architecture is a third-party component meeting all Jini requirements as well as providing an interface for the limited device. Consequently, two disjunctive aspects have to be considered:

- the relationship between the third-party component and the Jini-unable component, and
- the Jini behavior of the third-party component

The focus of this section, of course, is on the first point - the relationship between the components. Moreover, the basic Surrogate Architecture aspects are usually independent of each other, and thus the Jini Surrogate Architecture does not contradict the plug-and-participate model of Jini: It defines (Jini-compliant) rules for participating in Jini even if a certain device does not completely fulfill the Jini requirements. A third-party component, the so-called **Host Capable Machine** participates in a Jini network. It downloads and executes Java classes, and is able to act on behalf of the device. Such Host Capable Machines provide an appropriate execution environment in which the device's representative can be executed. This representative is called the **Surrogate**, while the execution environment is called the **Surrogate Host**. The Surrogate and the device itself are connected through a so-called **Interconnect**, which provides both the physical and logical connection. Via this link both parties - the Host Capable Machine and the device itself - first “discover” each other (via a Surrogate Architecture specific discovery protocol). The second step is the transfer of the Surrogate to the Surrogate Host and its execution there. Finally, the (application-specific as well as Surrogate Architecture related) communication between the device and its Surrogate is realized using a private protocol on the Interconnect:

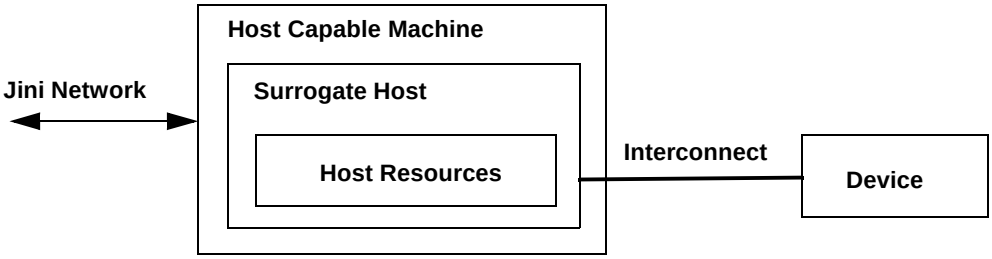


FIGURE 2. The basic structure and components of the Jini Surrogate Architecture (from [97], figure 1.1)

In the following sections, the architecture and the relationships of the components, and thus a general behavior pattern, are sketched. The main focus herein is on the Surrogate Architecture related behavior as well as on the connection to the Jini world.

Discovery

The first step in participating in Jini is discovery. In Jini, this refers to discovering LUSs, while in the Surrogate Architecture a discovery protocol is defined that supports the establishment of a connection between the Jini-unable device and the Host Capable Machine.

Remark: This discovery protocol does not refer to the Jini counterpart due to the complete independence of both Jini and the Surrogate concept.

Such connections, however, can rely on a wide variety of communication technologies such as standard Ethernet, bluetooth and fieldbuses. All these different technologies are considered by the Surrogate Architecture discovery protocol, which mainly defines the means of this connection rather than a particular implementation. This leads to Interconnect-specific discovery protocol implementations that strongly rely on the capabilities of the Interconnect as well as the device itself. For example, a device might broadcast for suitable Surrogate Hosts, or a Surrogate Host might poll the network in order to detect new devices arriving on the Interconnect.

Surrogate Upload and Initialization

Once discovery is successful, and a connection between the Surrogate Host and a device is established, the Surrogate must be installed at the Surrogate Host so that it can represent the device in a Jini system - it must be retrieved and activated. Similar to the discovery protocol, the Surrogate Architecture only defines a framework for this retrieval process because it also depends on the capabilities of the Interconnect and the device features: Either the Surrogate is pushed to the Surrogate Host (the device uploads the Surrogate) or it is pulled by the Surrogate Host (the Surrogate Host downloads the Surrogate either from the device itself, or from a code base). The Surrogate is subsequently initialized by the Surrogate Host so that it is executed at the Jini-enabled device. All Surrogate entities therefore must implement the `Surrogate` interface comprising the `activate(..)` and `deactivate(..)` methods. They are called by the Surrogate Host in order to control the behavior of the Surrogate. This means, a Surrogate Host generally handles all Surrogate objects uniformly through that interface regardless of the particular functionality and behavior of the Surrogate.

Representation of the device in the Jini infrastructure

A Surrogate represents its associated device, and thus it performs the necessary tasks on behalf of the device. This specifically includes access to the Jini network and the use of resources provided by the Surrogate Host. For example, the Surrogate might represent the device's functionality as a Jini service, or it can access the Jini community as a Jini client on behalf of the device. Both behavior patterns, client and service, require communication with the device that is provided using the Interconnect. If the Surrogate behaves as a Jini client, it has to forward the results of the function use to the device, while a service provision is defined by forwarding received requests to the device and finally to return the results. This communication between the Surrogate and its accompanied device is

obviously application specific, and thus it relies on a private protocol. Such private protocols, however, are neither covered by the Surrogate Architecture specification nor defined by the Jini technology:

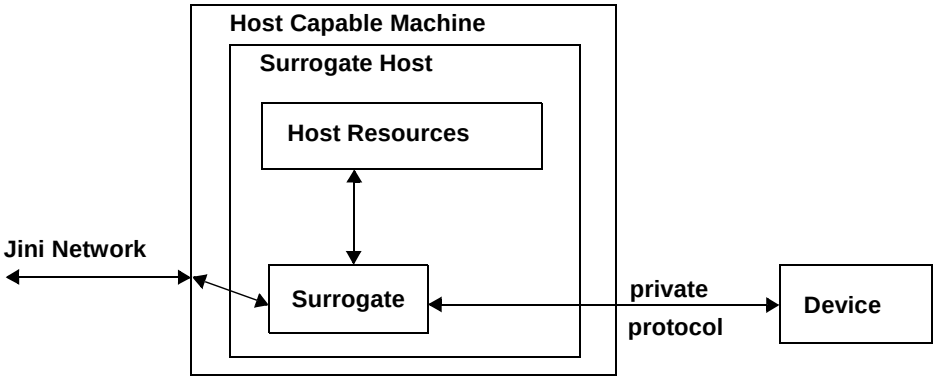


FIGURE 3. The final behavior of the Surrogate as the broker between the Jini world and the device

In addition to this basic infrastructure of the Surrogate Architecture, a mechanism for robustness must be incorporated into the relationship between device and Surrogate Host: A `Liveness` interface is defined that forces either the device itself or the Surrogate Host to monitor the relationship; i.e., whether the communication path between them is still active. If a device is no longer reachable for whatever reason, its Surrogate must be deactivated and all occupied resources need to be released. Likewise a device must be able to determine whether the communication to the Surrogate Host is broken. In this case, of course, the device should resume the discovery process, or otherwise cause the upload of a new Surrogate to a Surrogate Host.

In summarizing the Surrogate Architecture, the third-party component can be identified as the crucial point of this architecture. Such a component can directly participate in the Jini technology. The device itself is represented by a software entity, the Surrogate, which communicates with the device via a private protocol. Relying on such a third-party device increases the complexity of applications, which requires additional effort in terms of time, equipment, and thus money. In the field of industrial automation this eventually leads to a decrease in benefits due to decreased profit. This makes it rather unlikely that, in the end, Jini or the Jini Surrogate Architecture will be used in the field of industrial automation. However, the Jini concept in general and the defined interface suites are suitable for use in automation, but require a re-design of fundamental concepts like service use, service provision and attribute mechanisms. A technology having such re-designed facilities is specified in chapter 6, while chapter 5 shows an example of Jini in a plant automation solution. This solution, however, cannot be applied to automation devices directly due to the listed problems.

2.3.3 Universal Plug and Play

UPnP describes another important plug-and-participate technology available on the market. In contrast to the Jini technology, Universal Plug and Play (UPnP, see the specification under [56]) belongs to the category of technologies that do not rely on any central instance: It defines a completely different architecture than Jini, specifically the mechanisms utilized to provide the needed flexibility differ. Pro-

viding such flexible and adaptable networks and infrastructures of entities requires common mechanisms (defined in the abstract plug-and-participate interface in section 2.3.1) generally having the same objective as their Jini counterparts, but a completely different realization. This section, therefore, shows UPnP in a comparison with Jini and will also investigate the possible use of UPnP in the field of industrial automation.

First, a general overview of UPnP will be given, followed by a detailed description of the particular protocols. This detailed description of UPnP features closes the presentation of classic plug-and-participate technologies, and the focus is moved to their evaluation: Both technologies are investigated regarding their requirements in section 2.3.4.

2.3.3.1 UPnP Overview

The general motivation for UPnP, of course, is the same as for Jini: It was designed to provide easy-to-use, flexible and adaptable connectivity mechanisms allowing an alleviated establishment of communities. These mechanisms basically support networks in building infrastructures in a spontaneous fashion, so that UPnP can be abstracted as “...an architecture for pervasive peer-to-peer network connectivity..." of devices (see [56], section "Introduction"). The UPnP architecture therefore provides a distributed, open networking concept. It is heavily based upon TCP/IP and web technologies, which allows for transparent and seamless networking facilities. Such features are designed for a wide range of devices from different vendors. UPnP, in the end, enables devices for dynamically join networks, obtain IP addresses, offer their capabilities, and request the capabilities of other devices. A device also can leave a network smoothly and in a way that the system as a whole is not affected.

These previously described features are in conjunction with the basic plug-and-participate facilities of the abstract interface. Moreover, UPnP is provided as an abstract protocol rather than as a concrete API. UPnP mainly relies on a combination of already-existing protocols and technologies/standards. This integration of well-known standards provides the required plug-and-participate capabilities in the end - UPnP does not define “yet another” protocol suite. Likewise, it does not model its features as a monolithic protocol but instead provides them as a set of sub-protocols regardless of any programming language and (communication) media. The sub-protocols cover the two main aspects of plug-and-participate, namely the client-related aspect defining access mechanisms for functions and the device/service-related aspect specifying patterns for function provision. Strictly speaking, the UPnP protocol suite can be divided into three distinctive parts: pure functionality-related facilities like description and presentation, function access related features and finally protocols used by both, function providers and clients like IP address retrieval, discovery and “eventing”: For example, clients as well as devices have to obtain an IP address first, unless one is already assigned - UPnP strongly relies on IP addressing in order to identify and address entities in the community. Finally, each entity has to perform the so-called discovery protocol. This protocol either

- advertises a device and its respective services to the network, or
- searches for services of interest in case of clients.

Although the same discovery protocol is used by clients (called control points) as well as function providers, different behavior of the protocol is required. It basically leads to the exchange of different discovery messages: Function providers (devices) deliver discovery messages in order to announce their availability and their functionality, while clients request the infrastructure for devices/services of interrest.

Using the “discovered” information about existing facilities, control points gain more detailed knowledge of devices/services via the description protocol of UPnP. Such a description finally enables a control point for accessing a device/service, e.g., invoke actions, request variable states, request for notification if variables change (i.e., subscribe to events). All these interactions, of course, require certain service behavior such as the reception of service method invocations, subscriptions for events as well as publishing changed values of status variables. Devices and services, in turn, must provide suitable mechanisms satisfying such client requirements - they have to respond to variable requests and method invocations. They must furthermore provide their presentation and description to the infrastructures.

In addition to these rather service - service user related behavior patterns, devices/services can also benefit from the presentation facility. For example, devices can provide web pages offering useful tools like remote control interfaces, reporting or even monitoring and diagnostic facilities.

A comparison of the UPnP protocols with the Jini facilities shows abstract similarities between them - at least from the general plug-and-participate protocol view. Both concepts provide mechanisms for joining a community, request for services of interest, access the services, participate in a passive information system, etc. However, taking an under-the-hood view, an essential difference gets obvious concerning the general view to entities and infrastructure: While Jini specifies services and clients as basic entities, UPnP defines devices as the fundamental entities. UPnP devices, in turn, are characterized by a three-level hierarchy consisting of root-devices at the top level, (sub-)devices in the middle and services at the lowest level. Such a hierarchy implies three particular views of the infrastructures (see the specification in [56], section 1.2):

- **The root-device view**
  A control point only focuses on root-devices, regardless of their deeper structure.
- **The device-oriented view**
  Only devices are of interest, independent of their affiliation to root-devices.
- **The service view**
  Only services are considered independent of their specific grouping to respective devices or root-devices. This view, of course, is the most abstract one, because users are generally more interested in a certain functionality than in a specific device. Moreover, it is in strong correlation with Jini, which is simply service-oriented rather than device-based.

In order to clarify the different views, the hierarchy is shown in the following figure:

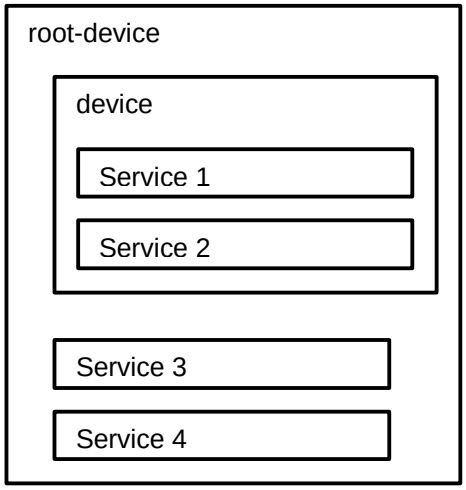


FIGURE 4. The hierarchical view of UPnP entities

The previous overview of UPnP will be the basis of the following section, which provides a detailed view of the UPnP protocols. However, only the most important facilities of the protocols will be tackled - if more information is required, the UPnP specification and related documents are recommended for study.

2.3.3.2 UPnP Protocols

The major objective of UPnP protocols is to ensure the basic plug-and-participate facilities but not to introduce a “new protocol suite” - UPnP mainly relies on the combination and utilization of already existing standards like TCP/IP, SOAP, HTTP, etc. This section therefore sketches the features of the UPnP protocol suite and finally points out the mechanisms and protocols it uses.

Addressing

The identification of UPnP entities is based upon the Internet Protocol (IP), and thus each entity must possess an IP address. For this address retrieval process, UPnP entities either use DHCP or Auto-IP (see also [79]) - the preference lies with DHCP. However, if no DHCP facility is available, the entities must use Auto-IP for address retrieval, but must periodically check whether a DHCP service becomes available.

Remark: Auto-IP means the selection of an IP address, and the subsequent test whether this address is already in use (via ARP, see [79]).

In this case, the address obtained via DHCP must be used. Such a behavior furthermore makes it reasonable to publish only the (host) names of devices instead of IP addresses because the addresses might change in contrast to their names.

Discovery

UPnP entities use the discovery protocol in order to form ad-hoc networks - devices must offer (advertise) their availability. This allows client entities to obtain knowledge about the community: As an architectural fundament, devices MUST send their advertisement messages to the community, while clients CAN listen for those advertisements. The technological prerequisite is the Simple Service Discovery Protocol (SSDP). It defines the use of multicast communication on address 239.255.255.250, port 1900 in order to deliver the advertisement messages. These messages, moreover, are also specified by SSDP, which internally utilizes specific multicast and unicast variants of HTTP for message transmission (see therefore the SSDP specification [38]).

Based upon these architectural and technological basics, UPnP defines the required behavior patterns for advertisement and function search: The advertisement messages must be sent corresponding to the hierarchy shown in figure 4. In particular, a root-device sends three messages for itself, two for each embedded sub-device, and finally one message for each provided service via multicast to the network. All these messages are only valid for a certain period of time, and thus they must be periodically re-sent. Such validity periods must be nearly the same if they are related to entities of the same device - this ensures the essential robustness of UPnP communities, because unavailable or non-functional entities are removed automatically (compare this behavior to Jini's lease mechanism). The periods of the advertisements are proposed to be rather long in order to decrease the network load because they are sent via multicast, which causes high network load so that short intervals can exhaust network bandwith.

Taking this basic robustness behavior into account and assuming a certain fluctuation within such networks, entities require an efficient way to leave a community properly: The UPnP discovery protocol furthermore defines advertisement messages that are used to revoke the availability of devices or services.

The second aspect of the UPnP discovery protocol aside from the advertisement and revocation of entities is the function search, which is usually performed by control points: In the same way as devices, they send messages via multicast to the entire network (address 239.255.255.250, port 1900). Such search messages carry suitable search patterns or descriptions of the targets. Devices, in turn, must listen on the aforementioned port. If they can fulfill a request, they respond with an advertisement message that is the same as used for the general advertisement facility. However, this response message, in contrast, is sent via unicast to the requesting control point instead of responding via multicast.

Description

Advertisement messages only inform control points about the availability of particular services - they do not provide any details pertaining to service access, such as interaction descriptions, method signatures, variables, etc. In contrast, interested control points have to request this information via the UPnP description mechanism, which is a two-stage process. First, control points request the device description, which provides the information regarding the physical and logical containers of a device. This information is vendor-related such as model or serial number, but also more general information (such as a URL) for detailed service descriptions is provided. With the help of this service information retrieved from the given URL, a control point can access a particular service, e.g., invoke methods or request status variables (see [56], section 2.1/2.3).

The UPnP description mechanism also follows the paradigm of simplicity, generic makeup and utilization of already-existing standards - the descriptions are XML-formatted and are usually retrieved via HTTP `Get` requests. They are also annotated with a certain validity period that corresponds to that of associated advertisements. If the corresponding advertisement is revoked or expired, its descriptions also become invalid and are subsequently discarded. If this did not occur, the robustness, stability and reliability of the system would no longer be ensured.

Control

One of the main aspects of plug-and-participate systems is the guaranteed transparent access to the provided functionality. UPnP basically involves the invocation of methods as well as the request for service status variables defined in the service descriptions. In keeping with the paradigm to utilize already-exising standards, UPnP relies on SOAP for those interaction patterns. In particular, "SOAP defines the use of XML and HTTP for remote procedure calls" (see also [56], section 3.2.1]). UPnP control messages are therefore delivered via SOAP, i.e., method invocations and their respective results. Such method invocation messages are XML-formatted and transmitted via HTTP `Post` requests. For example, a UPnP method invocation based upon SOAP might resemble the following code example:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="some-URI"
                            SOAP-ENV:encodingStyle="some-URI">

  <s:Body>
    <u:actionName xmlns:u="urn:schemas-upnp-org:service:serviceType:v">
        <argumentName>
             in arg value
        </argumentName>
             …
             other in args and their values go here, if any
             …
    </u:actionName>
  </s:Body>
</SOAP-ENV:Envelope>
```

CODE EXAMPLE 10. A UPnP action invocation (from [56], section 3.2.1)

After the message has been sent out, the requested service has to respond within 30 seconds. If an action takes longer, it must return earlier. This means, the method returns a certain result following the SOAP XML RPC pattern. Such a reponse might look like the following code example:

```
<s:Body>
  <u:actionNameResponse xmlns:u="urn:schemas-upnp-org:service:
                                                    serviceType:v">
       <argumentName>
          out arg value
       </argumentName>
        …
        other out args and their values go here, if any
        …
  </u:actionNameResponse>
</s:Body>
```

CODE EXAMPLE 11. The response of a UPnP action invocation (from [56], section 3.2.2)

The final results of interactions are stored in the service status variables. These variables can be queried using the following SOAP request - with one request per variable:

```
<s:Body>
      <u:QueryStateVariable xmlns:u="urn:schemas-upnp-org:control-1-0">
           <u:varName>
              variableName
           </u:varName>
      </u:QueryStateVariable>
</s:Body>
```

CODE EXAMPLE 12. A UPnP request for a certain variable (from [56], section 3.3.1)

The queries for variables also rely on the request - response pattern, as is usual for method invocation. In particular, a requested entity has to respond within 30 seconds delivering the variable:

```
<s:Body>
   <u:QueryStateVariableResponse
                          xmlns:u="urn:schemas-upnp-org:control-1-0">
      <return>
        value
      </return>
   </u:QueryStateVariableResponse>
</s:Body>
```

CODE EXAMPLE 13. The response of a variable request (from [56] section 3.3.2)

This process of requesting variables describes the active fashion of information retrieval in UPnP. However, if a control point is interested in changed variables, i.e., variables changed due to formerly invoked UPnP actions that returned earlier than the action has finished, it can subscribe to events reporting such changes. The next section describes this process.

Event Facility

If a control point is interested in being informed about any changed variables of services, it must register with the services for that purpose. This means, it must send a subscription message to them.

Remark: UPnP's event facility is based upon the General Event Notification Architecture (GENA, see also [17]).

A service, also called publisher in this respect, must respond with an unique identifier, if it accepts the subscription. Furthermore, the publisher must send an initial message containing the names and current values all reported variables. Such a message is sent to a URL that has been specified within the subscription request, which furthermore consists of a service identifier and a validity duration. This duration will be acknowledged by the service, and the control point must periodically re-send its subscriptions in order to renew the registration. When a service changes its variables (and if the affected variables are reported) it sends an event message to all registered control points - there does not exist any filter mechanism for control points for specific variables - control points receive event notifications for all reported variables.

Finally, in correspondance to subscription messages, UPnP's event mechanism also defines appropriate (GENA based) unsubscribe messages in order to leave the event mechanism. If a control point transmits an unsubscribe message, the addressed service must acknowledge the request within 30 seconds.

Presentation

The presentation facility is not directly concerned with the provision of dynamic and spontaneous service networks, but basically allows devices to offer their capabilities to interested parties in a generic and standardized fashion. For example, a device might provide a web page or similar facilities that can be accessed by control points, in order to obtain additional information about the service/device - they may provide a URL in their description, so that control points can access this presentation facility. Likewise, following the paradigm of utilizing existing web technologies, UPnP's presentation facility is based on HTTP and HTML, and thus entities can provide information or remote control interfaces through arbitrary web pages.

2.3.4 Jini and UPnP Requirement Evaluation, Comparison and Inherent Drawbacks

Using the knowledge about the technical and architectural details of classical plug-and-participate technologies their requirements must be evaluated and then compared. This generally covers the facilities defined by the abstract plug-and-participate interface and the particular implementation within Jini and UPnP. Further aspects considered herein can be derived from the particular architecture, and those resulting from the behavior of the technologies. For example, both technologies provide platform independent concepts. UPnP, moreover, is also programming language independent, while Jini strongly relies on the Java programming language. Both approaches show advantages and disadvantages: Jini can directly take advantage of Java features like security and dynamic remote class load-

ing, and thus is tailored to Java. Especially security issues require the use of Java 2, Standard Edition or Micro Edition CVM/CDC. One disadvantage is that Jini cannot directly be applied to other programming languages without architectural adaptations. This basically limits Jini's flexibility and field of application. UPnP, in contrast, benefits from its openess to almost all programming languages - it allows a broad field of application and an alleviated realization for different environments ensuring a high degree of flexibility. A further architectural aspect refers to the necessity of a central instance for infrastructure management versus a completely distributed infrastructure: Jini's LUS as a central instance defines a single-point-of-failure, and thus impacts the reliability and robustness, while UPnP's reliability and robustness strongly relies on a high degree of redundancy and network communication. This redundancy and network load results from the protocols and mechanisms used by UPnP - it will be evaluated in the comparison of technological facilities.

The fourth architectural focus pertains to the goal of UPnP to rely on already existing and thus established technologies, protocols and web facilities instead of defining simply a "new protocol suite" - Jini defines its own mechanisms to provide plug-and-participate facilities. It therefore benefits from the possibility to implement such protocols in a rather customized fashion, and it uses proprietary (and thus customized) mechanisms. There is only one exception - Jini defines the use of the Java RMI standard for its remote event facility. UPnP, however, relies on established and standardized protocols like IP, SOAP, SSDP, HTTP, XML RPC, etc., and therefore depends on their availability on the target platforms, their particular implementation and their specific facilities. A direct consequence of this is that UPnP is strongly bound to IP and the address retrieval via DHCP or Auto-IP. Jini, in contrast, is open to almost all network technologies, and thus can also be applied in non-IP environments.

After these rather general requirement aspects, the focus is moved to the technological requirements. The basic aspects to be measured and evaluated are defined in the abstract plug-and-participate interface:

Community Join

The community join process defines the means by which entities offer their capabilities to the network. In UPnP, the SSDP protocol is used - each UPnP entity that intends to offer functionality has to send SSDP-compliant advertisement messages - three for a root device, two for each embedded device and one for each embedded service. All of these messages are sent via multicast, as defined in SSDP.

Jini's join process is a two-stage process: First, at least one LUS reference must be discovered before a dedicated join message can be sent to the LUS. The discovery process depends on the protocol used - if the Jini multicast discovery protocol is used, seven messages will be sent via multicast to the network. When such LUS references are obtained, the service sends one dedicated join message to each LUS.

Service Search

As the complement to the community join, each plug-and-participate technology must define a service search facility. UPnP relies on the SSDP protocol, which specifies the transmission of dedicated search messages sent via multicast to the network. The result is the knowledge about devices and their services. Likewise a URL is provided, where detailed access information can be retrieved via UPnP's description mechanism using HTTP.

Jini, in contrast, assumes the availability of a LUS reference that is used to send a lookup request via unicast to the LUS. If matching services are found, their references (service proxies) are transmitted to the requesting client.

An important aspect in this respect is the matching of requests for available functionalities - Jini relies on a three-stage process, while UPnP's matching is based on certain SSDP facilities. Jini's three-stage process, compares the Jini `ServiceID` of a request with that of registered services, or matches the Java types of a request to registered services, or compares the attribute sets. This attribute comparison, however, relies on the comparison of `MarshalledObject` representations of the atttributes, and thus the attribute byte representation - only exact matching is ensured.

SSDP search messages, in contrast, are XML-based. They specify patterns for searching for

- all entities in the community,
- specific service types,
- dedicated services addressed by their UUID,
- root devices
- device types.

Thus, despite the currently implemented Jini matching algorithm allowing only exact matching, its search facility can be considered more flexible and efficient than UPnP's counterpart - the Jini search mechanism allows a more detailed filtering than UPnP. An improvement of Jini's matching facility is shown in more detail in section 5.3.3.3.7.

Robustness Facilities

In order to satify the robustness requirement, UPnP entities periodically have to re-advertise their availability - they periodically re-submit advertisement messages via multicast to the network. Each advertisement is annotated with a period of validity that allows entities to track whether an entity is still available. Jini uses a similar mechanism called leasing - each reserved resource in Jini is annotated with a lease that must be periodically renewed. This renewal, however, is done per unicast with one renewal message per lease and period. Such a mechanism obviously conserves network resources: Although a service can be registered with several LUSs, only connections to affected LUSs are established. UPnP, by comparison, floods the network with advertisements and subsequently forces every entity to process these messages even if they are not of interest. The network and entity load can only be decreased by prolonging the validity period, which in turn compromises the up-to-dateness of the community.

Information Flow Facilities

Jini's information flow facility strongly relies on its central instance, which has an always-up-to-date overview of the entire community. If entities are interested in information about events in the infrastructure, they must register with the LUSs by specifying events of interest in a detailed fashion. Event dispatching in Jini is RMI-based, because the local Java event model is extended to a distributed model. A LUS simply invokes the `notify(..)` method of the listener object - it is remote via Java RMI. This ensures a standardized view to the event model. The registration for remote events is centered on leasing, as usual in Jini.

UPnP's GENA-based mechanism also relies on the registration of interested parties, but, in contrast to Jini, they cannot specify dedicated events of interest. In the same way as in Jini, UPnP events are delivered via unicast communication to the interested entities. Each entity gets informed about each event occuring at a certain device, where the entity is subscribed. Of course, this increases the network load as well as the load of the devices themselves: Services usually have many of status variables that may rapidly and frequently change. If an entity is only interested in a few variables, it nevertheless receives an event if even a variable changes in which the client is not interested - this increases the network and entity load unnecessarily.

Service Use

The most important fundament in plug-and-participate systems along with ad-hoc networking is the use of participating entities, namely services. Jini and UPnP define completely different mechanisms for that purpose: While Jini relies on its proprietary proxy concept, UPnP defines the use of SOAP and XML RPC as the basis for service use.

The Jini service proxy concept allows entities to transparently access the service - the proxy behaves like the service and usually also forwards the request to the service. Unfortunately, this concept relies on the execution of foreign code on hosts, and this in turn requires class downloading and strong security policies (for more information on such security aspects please refer to [90]). The major focus here is on the class download requirement: If a proxy refers to many classes, they need to be loaded in order to execute the proxy. In turn, if these classes themselves refer to further classes, they also must be loaded. This obviously results in an unpredictable avalanche of classes that may exhaust memory. Especially in certain fields like automation as well as limited devices, such unpredictability is impermissible in order to ensure stability, safety and security, etc.

SOAP, by comparison, simply relies on the exchange of XML-formatted messages that do not require any class loading: If a method is to be invoked, a dedicated message is submitted (see code example 10) to the addressed object via SOAP. However, using XML usually requires a suitable XML handling mechanism, i.e., an XML parser. XML handling, in turn, is generally rather resource-consuming, and thus might not be suitable for limited devices. Moreover, the mechanisms also have a certain data overhead due to tags enriching the data.

Among these rather technological and resource aspects, this XML RPC pattern also lacks transparence - objects simply exchange messages defining the method/interaction to be invoked as well as the accompanied parameters.

Finally, the resource consumption of the technologies will be discussed, which mainly covers an evaluation of the memory consumption on disk and during runtime. For Jini, the disk space of the Jini framework is measured that will be shown in table 1 in more detail. The memory consumption during runtime depends on the measured entity: While a LUS often consumes more than 10 MB RAM, a Jini service that simply provides a proxy to the network and furthermore performs the standard Jini protocols, normally comsumes up to 10 MB. Measuring client requirements is rather difficult because the resource consumption strongly depends on the services used (remember the unpredictability of classes to be loaded), and thus, no exact data can be provided here. The same arguments apply for UPnP - its resource consumption essentially depends on its implementation and the implementation of the required protocols. Therefore, no exact measurements can be provided for that purpose.

All these aspects will finally be aggregated in the following table which summarizes the comparison of the requirements, made in this section.

Facility	Jini	UPnP
platform	platform independent	platform independent
programming languages	Java-based, at least Java 2 Standard Edition or Java 2 Micro Edition, CVM/ CDC	programming language independent
basic architecture	uses centralized instance (LUS)	completely decentralized
protocols	not fixed to any protocol, execpt RMI for remote event facility	uses approved protocols like SOAP, IP, HTTP/HTML, XML, RPC, DHCP, GENA, SSDP
network prerequisites	open to almost all network technologies	IP-based
join mechanisms	first discovery of LUS references via multicast or unicast, transmission of a join messages via unicast consisting of `ServiceID`, service proxy and attribute sets	SSDP-based, XML-formatted advertisement messages sent via multicast
service search	first discovery of LUS references via multicast or unicast, transmission of lookup messages via unicast consisting of search patterns	SSDP-based, XML-formatted search messages sent via multicast, or tracking of advertisements
robustness features	leasing of resources, periodical renewal via unicast communication	periodical re-advertising of availability via multicast
information flow	registration with LUS, detailed definition of events of interest, dispatching via unicast	subscription to events, no possibility for event filtering, event dispatching per unicast
service use	service proxy concept requiring dynamic remote class loading, no fixed communication between proxy and service	strongly based on SOAP and XML RPC
communication facilities	unicast and multicast communication facilities required	unicast and multicast communication facilities required
disk space of framework	Jini core 31.7 kB Jini extension 266 kB LUS 573 kB LUS proxy 150 kB RMI 83.4 kB **Sum 1104.1 kB**	abstract protocol, thus no API-like framework, no exact data

TABLE 1. Jini and UPnP requirements

Facility	Jini	UPnP
runtime memory (central instance if existent)	>10 MB	no central instance
runtime memory - service	~10 MB	no exact data
runtime memory - client	depends on services used, no exact data	no exact data

TABLE 1. Jini and UPnP requirements

This comparison has shown strength and inherent drawbacks of both technologies. Their drawbacks might inhibit their use on limited-device platforms and automation devices: Jini's class loading necessity and the accompanied unpredictability of memory consumption, and UPnP's tremendous network load are the key factors in this respect.

CHAPTER 3

State of the Art in Industrial Automation

This chapter covers the second and third major facets of this thesis - the field of industrial automation as well as the field of limited devices. The first facet, plug-and-participate, has already been introduced in the previous chapter. Here the state of the art in industrial automation is presented, sketching the history, which starts with traditional centralized systems and shows the current trend towards distributed systems - an overview of automation and plant systems in general will be given in order to reach a common understanding of plant automation. A common model will be introduced, which is the starting point of discussions about distributed automation solutions. This model, in turn, is used in this chapter to show the general structure of enterprises in a layered fashion and to derive the motivation for distributed automation systems as the basis for discussion in chapter 5.

The evaluation of the state-of-the-art covers recent work already done in the field of industrial automation. With respect to the considered turbulent markets, the focus of this chapter (and likewise this thesis) is on distributed automation solutions such as the Holonic Manufacturing System (HMS) concept, the Interface for Distributed Automation (IDA), but also a distributed communication solution for industrial automation (PROFInet) is shown.

The third topic of this chapter is an evaluation of today's automation devices, especially their technological requirements in terms of computational speed, hard disk space, memory and networking facilities.

All three facets - plug-and-participate, plant automation, and target devices - will finally lead the reader to an automation paradigm and a solution based upon plug-and-participate that is also suitable for legacy automation devices generally having a limited-device character. Bringing these three facets together and building an inte-

grated solution is the major aspect of this thesis - concrete automation field related solutions are shown in chapters 5, 6 and 7.

3.1 The Automation Pyramid

The evolution in plant automation started in the very beginning of industrialization with the foundation of the first small industrial companies. Decision making and planning - the fundamental management facilities and basic plant structures - have not significantly changed since that:

- First "plants", which have been founded in the beginning of industrialization, were generally led by one owner - he was responsible for decision making and planning, and therefore a "central planning" system was established.
- The size of plants was growing steadily, and legal forms of companies have been introduced (see [5] and [40]), while the general structure of planning and decision-making processes remain centralized at appropriate central instances.

The increased size and complexity of plants, planning and decision-making processes led to a partitioning of these “intelligence centers” into distinctive parts forming the following hierarchy:

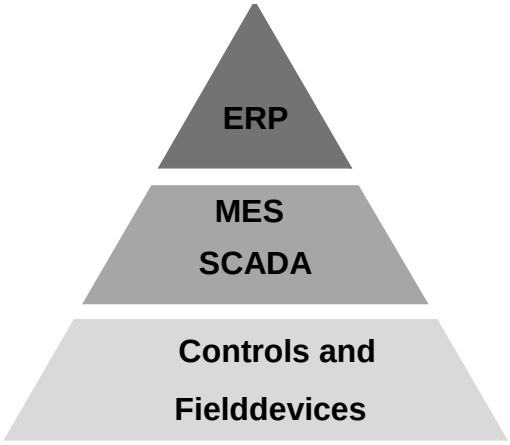


FIGURE 5. The CIM Automation Pyramid [81]

An Enterprise Resource Planning system (ERP) of a plant is generally responsible for managing the overall planning of the plant - it comprises sales as well as purchase departments and the resource planning. ERP systems basically acquire orders for manufacturing, create appropriate orders for production (with a certain degree of granularity) and plan required resources (Bill of Material, inventory in general, etc.) as well as the basic manufacturing process/flow. They are reasonably centralized, at least from the historical point of view and based upon their dedicated tasks.

At the subsequent level, so-called Manufacturing Execution Systems (MES) are situated. They are responsible for the detailed planning of the manufacturing itself and provide scheduling facilities,

resource allocation mechanisms, product tracking, etc. (for the different MES functionalities see [54]). Planning at this level is more detailed than at the ERP level. For example, it allows the maximization of plant throughput and also the minimization of production time and costs. However, optimizations like these basically require a detailed overview of the load and available resources at the plant. This leads to centralized MES systems suitable for mass production, flow production processes, etc., yet they have serious disadvantages in turbulent environments, e.g., if only small lot sizes of frequently changing product variants must be manufactured. The second facet of the middle level is SCADA - Super Visory Control and Data Acquisition. This facility is mostly used for monitoring and control of machines, and is therefore usually centralized.

The lowest level of the automation pyramid concerns the field level including field devices with their Programmable Logical Controllers (PLC), sensors, actuators, I/O channels, communication channels, etc. This field was originally characterized by a decentral architecture: Autonomous PLCs consisting of a central processing unit (CPU), partially centralized I/O modules and linked sensors and actuators. Such an architecture comprised independent and autonomous entities performing their own tasks. With the introduction of fieldbuses, the I/O modules moved closer to sensors and actuators, which in turn became more and more “intelligent,” but remain controlled by the associated central PLC programs - formerly autonomous PLCs turned more and more into centralized network nodes (see for a more detailed view [45] and [68], especially the motivating sections).

Such historically grown legacy architectures of plant automation systems are mainly based on centralization and concentration. Also, the automation pyramid is tailored to those systems - the layers, drawn as monolithical blocks, illustrate the “compentence centers” within plants - structures like these, of course, have their strength and their drawbacks: Advantages of centralized systems are in the concentration of knowledge, authority and decision/planning processes - they always have a complete overview of the entire system. Such a complete overview allows alleviated optimization towards increased throughput of the entire plant, and thus is accompanied by the optimization of the machine load of the plant - plants and their optimization are infrastructure-driven in order to achieve a maximized and balanced load of the plant. These facilities are moreover fundamentally tailored towards mass-production, where throughput and output of the system are the driving key factors. In systems, where frequently changing prerequisites, requirements and situations are the key factors, and thus a high degree of flexibility and adaptability in planning is required, optimized and centralized systems are not well-suited. For example, strongly customized products and small lot sizes do not require optimized or maximized plant throughput and machine load, but rather a compliant reaction to new demands reflected in further product variants - here flexibility and adaptability are the driving factors. The algorithms and general structures of central planning systems and central MES facilities are not able to cope efficiently with such problems: If changes occur, they may affect the entire planning, and thus a complete re-planning could be necessary. Complete re-planning, of course, cannot efficiently provide the needed flexibility, specifically due to rather time- and cost-intensive planning processes.

A further, but fundamental disadvantage of centralized systems lies in their centralized nature - they define a single point-of-failure. If a planning system fails, typically the entire plant system is affected, as well as if planning mistakes occur - this, too, may harm the entire planning (more details on this topic can be found in [41]).

The comparison of the strengths and drawbacks of centralized planning systems with the requirements of more and more turbulent markets shows an orthogonal fashion: The basic requirements of

such turbulent markets - flexibility and adaptability - are the shortcomings of current centralized planning systems. Satisfying these markets requires a re-design of the planning paradigm. A trend towards non-centralized planning and automation systems is reasonable. Moreover, today's markets are customer-oriented, and the crucial point for a successful business is the ability of vendors to "completely" satisfy the customer demands. It will (eventually) ensure the survival of an enterprise in the market - a natural evolution can be observed, and it is this evolution that will lead to distributed planning systems.

3.2 Existing Automation Solutions and Related Work

Centralized automation systems have been sketched in an overview - they obviously do not meet the requirements of today's turbulent markets. This section now concentrates on solutions and proposals for distributed automation systems, since the focus of this thesis is on automation systems that are able to deal with rapidly changing demands and surroundings - a kind of paradigm change can be observed (see also section 5.1), where the historically grown centralized systems will be replaced by decentralized structures. Such a paradigm change increases the survivability of companies in their markets and surroundings.

The automation solutions sketched here are the first efforts in making vendors more flexible to customer demands and trends. They will be the starting point for the development of a new automation system that relies on plug-and-participate technologies and agent facilities. Two of the presented solutions will be furthermore the main input for plug-and-participate-based proposals for field-level-related solutions, which are surveyed in chapter 5.

3.2.1 Holonic Manufacturing Systems

The need of distributed automation solutions recently arose because centralized systems are no longer able to efficiently cope with the stated turbulence of markets, certain efficiency requirements, rapidly changing customer demands, etc. However, the different levels of plant automation must be considered. For each level of that pyramid, specific solutions can be provided. One of the solutions for the MES level is the concept of Holonic Manufacturing Systems (HMS). This solution is described in several documents such as [10, 11, 12, 13, 14, 41] - the current section provides an extract out of these documents.

The HMS concept is based upon the autonomous behavior of orders that, abstractly seen, organize their manufacturing processes independently of each other. Each order is modeled as an order holon, which in turn determines the resources required for its manufacturing processes. These resources are represented by so-called resource holons. They are freely organized as manufacturing holons representing the basic level of HMS. An order holon, once it enters a manufacturing holon, takes the initiative and starts a search process for resource holons that meet the requirements given by the order.

This results in a certain (virtual) sequence of resource holons that abstractly behave like a manufacturing line:

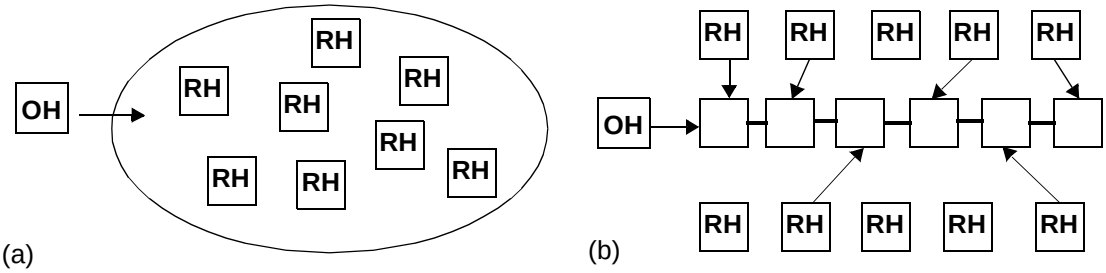


FIGURE 6. A virtual manufacturing line as a sequence of resource holons (from [13], figure 4)

During this preparation step, the order holon uses specific negotiation and communication mechanisms for that purpose: It requests available resources, compares offers with needs, etc. The resulting manufacturing line defines a nearly optimal sequence of resources; the basic strategy of those holons is to maximize their benefit. For example, each resource holon attempts to maximize/optimize its work load, while an order holon optimizes its processing and plans. After finalizing this first preparation of order processing, the order holon initiates the creation of a work piece holon representing the associated work piece. This new holon is responsible for controlling the product processing based on the provided resource sequence. Such controlling processes basically reflect scheduling and resource allocation steps in order to define the final path through the plant system.

When the order is finished, the previously allocated resource holons are no longer needed for that order, and thus the resources can be released. The manufacturing holon dissolves into its particular resource holons that in turn can form a new manufacturing holon, join other manufacturing holons, or simply disappear from the plant system.

The entire concept described here shows dedicated plug-and-participate patterns: Resources are freely organized in a community, arranged in a certain sequence, and finally dissolve into their single parts so that new "resource holon communities" can be built.

An important aspect that can be derived from this overview is the two-layered architecture of a holon: the information processing layer and the physical processing layer.

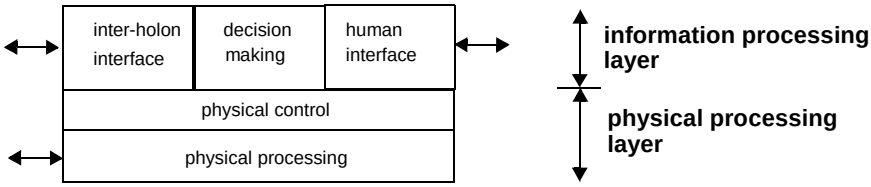


FIGURE 7. The layers of a holon (from [12], figure 1)

The physical processing layer can be assumed as rather static, since it simply provides the automation functions for the physical processing. Fundamental HMS operations are mainly concentrated in the

information processing layer, and thus the main focus is on this layer. Within these components all its logical operations and decisions are made. For example, the simple use case sketched the main activities of holons: negotiation, scheduling and resource allocation, but also the provision of resources, capacities and capabilities. This behavior is a reasonable consequence of the major focus of HMS - planning and MES processes rather than manufacturing processes in general. Such tasks are done autonomously by each holon in order to reach the local goal of maximized benefits. The global goal, in contrast, requires the ability of holons to cooperate. This necessary cooperation and interactivity is provided by the HMS concept through suitable (inter-) holon interfaces as well as particular interactivity and communication patterns. Taking these two basic, but contrary facts together - the autonomy of holons and their need to cooperate - and moving the view towards the implementation of the concept, one class of technologies in particular easily fulfills these requirements: agent technologies (for this design decision also refer to [12]). Agents are usually autonomous entities when performing their tasks, but can also be cooperative in order to achieve a global goal by following their particular strategy.

3.2.2 Interface for Distributed Automation

Accompanied with the introduction of new solutions on the MES level, also the field level requires flexibility and adaptability in order to be compliant with the improved upper level. The most notable fact is that the structure of (field level related) applications must follow this necessity, and therefore, solutions able to cope with changing surroundings are required. One proposal in this respect is the Interface for Distributed Automation (IDA), which offers a design concept for automation applications. It is based upon a hierarchical structure as well as a set of functions (see the IDA specification under [45]). The main focus of IDA, in terms of application creation, design and engineering, is on the requirements of the application to be created rather than on technological prerequisites of underlying devices - the key factor of IDA can be abstracted to the semantic of applications rather than the physical functionality of devices. Herein an important innovation in the automation field is obvious because an application as the controlling entity is usually more important for the efficiency of a process than the physical functionality itself. A reasonable compromise between the functions and the application logic must be found in order to realize the goal of an efficient system. IDA therefore does not exclude the physical level from its considerations, but instead incorporates the physical functions as abstracted entities. This leads to a two-layered general architecture: The runtime layer, comprising the automation functionality, and the engineering level, taking on the application. Both layers are defined in specific views. The topological view of IDA abstracts the underlying physical topology to IDA-compliant objects, and thus offers a set of functions to the functional view. This view, in turn, defines the formal guidelines for building an IDA application. The result of these definitions is the IDA System Model consisting of five sub-models: Process Model, HMI Model, Presentation Model, Engineering Model and finally the Application Model covering all other models. The Application Model, together with the functional view, is the crucial aspect of IDA. Both together define the components of an application as well as the relationships within the system. These components are the application itself as the top level of the architecture, sub-applications that are nothing other than applications themselves, and finally the IDA Blocks abstracting the basic automation functionality. Applications usually comprise subapplications, which in turn can either define further subapplications or finally IDA Blocks. The resulting recursive structure allows the modeling of almost all applications of any complexity. Migrating this logical application structure to the physical infrastructure of the underlying automation functionality connects both layers of IDA, namely the engineering and physical layer. Physical functions are abstracted as a set of independent functions, which

are topological view objects such as so-called IDA Resources. The IDA Blocks defined in the functional view are mapped to these IDA Resources, and thus the connection of the logical application to the physical automation world is established. This mapping, however, is defined rather statically by appropriate catalogue information. For example, appropriate element type information are offered, which are used for the mapping. IDA can therefore receive an important improvement if this mapping functionality is automated - for instance, by suitable plug-and-participate mechanisms. Such a proposal is described in section 5.4.2.

3.2.3 PROFInet

PROFInet, defined in [83] and further evaluated in [81], describes a second field level automation solution that, in the same way as IDA, concentrates on the definition of a new standard for automation applications. However, in contrast to IDA, where the application and its particular semantic is the driving factor, PROFInet tailors applications for particular devices, and thus devices are the key factor. Despite this essential difference between both technologies, PROFInet also follows the trend of defining patterns for distributed automation application creation. The PROFInet architecture is similar to the IDA structure - it also defines a two-layered architecture: The runtime layer representing the physical automation level by automation objects, and the engineering level comprising so-called engineering objects. Both object types are linked together by PROFInet specific connections that are manually set. This is accompanied with the manual download of necessary connection information. Corresponding to this two-layered architecture, and also to the fundamental device-orientation of the PROFInet concept, an abstraction of the engineering layer is required. Moreover, also the more device related runtime layer must be abstracted. This results in objects of the respective levels, namely engineering objects and runtime objects. Both types are furthermore divided into three runtime objects and three engineering objects. For example, the runtime system describes a physical device object as the root of the object hierarchy. Such a device (object) can contain several logical devices, which in turn can be linked to one or many runtime automation objects representing the physical functions. This structure obviously spans a tree, whose nodes are mapped to engineering view objects defining the logic of the application. These engineering view objects modeling the application on the logical level must be in correspondence to the runtime automation objects: ES-PDev for the physical device, ES-LDev for logical devices, and finally ES-Auto for runtime automation objects.

Additionally to this basic architecture, its basic components and their general relationships, also the interaction relationships within those object systems defined by PROFInet or IDA must be taken into account: While IDA does not define any specific technology for that purpose, PROFInet specifies the use of COM/DCOM, TCP, UDP, IP, and Ethernet. The reasons for this can be found first in the intention to apply broadly-accepted standards from the IT world and to use proven and well-known technologies. A second reason for this is that the mentioned standards are rather old, and therefore changes usually occur rather seldom. Especially the use of such outdated technologies as COM and DCOM for communication at the physical level, and specifically for communication between the automation objects can be motivated in this concern. Stable, safe and secure applications are an important factor in plant automation, due to the possible risks in terms of undesired behavior in components and the possible dangers for personnel and equipment that could result from a malfunction. Also financial risks in terms of unprofitable (due to unstable) solutions must be considered.

3.3 Intended Target Devices and Platforms

The application of plug-and-participate based automation solutions to legacy automation devices will lead to the third major facet of this thesis - namely limited devices. Legacy automation devices are evaluated within this section in order to determine their requirements, especially the resources and platforms they provide. Depending on those platforms, the design constraints for the automation solutions and, most importantly, for the plug-and-participate concept can be derived. Moreover, these legacy devices are the basic test platforms on the one hand, but on the other hand they also define the initial application range of the concepts that will be shown in chapter 5 - the application of new concepts to legacy devices is a crucial criterion for a broad acceptance of new concepts.

3.3.1 State-of-the-Art Automation Devices and Platforms

An evaluation of state-of-the-art automation devices not only considers devices on the plant floor level, but generally reflects the entire automation pyramid, and therefore all levels in a plant: The office level as well as the MES level are mainly equipped with PC-based hardware due to the execution of rather management-related tasks. The field level, however, differs from the upper levels because it basically shows a heterogeneous device structure ranging from PC-based hardware and controller hardware to fieldbus devices like sensors and actuators.

PC-based Hardware

PC-based hardware usually comprise standard PCs equipped with modern components, e.g., 3.5 GHz processors, ~ 100 GB hard disk and powerful graphical adapters. Such devices run on standard operating systems and Java platforms like the Java 2 Standard Edition.

PC-based hardware at the field level differs slightly from a simple office PC - industrial PCs (IPC) are customized for the requirements of the field, and thus they are usually some steps behind the evolution. This is reasonable, because the focus on the field level is more on security, safety, reliability and robustness rather than on performance.

Controller Hardware

Controller hardware mainly covers the field of PLCs. There is a distinction into soft- and hard-PLCs: Soft-PLCs are programs representing the controller logic but do not require any dedicated processor - e.g., they use the resources of an arbitrary PC. Hard-PLCs, in contrast, run their control applications on a dedicated processor, or the functionality is implemented in hardware. Resources provided by hard-PLCs range from rather powerful devices down to resource-constrained devices as shown in the following table 2.

Remark: The devices listed in the table are those used by partners of the PABADIS project, namely Jetter, Phoenix Contact, and the Institute for Automation and Manufacturing (IAF, Institut für Automatisierung und Fabrikbetrieb). These devices can be seen as representative state-of-the-art devices.

Product	CPU	Vendor	Speed (MHz)	RAM (MB)	ROM flash/ (MB)	Network Adapter/ Ethernet (MBit/s)	OS
JetControl 241	NetArm 40	NetSilicon	33	4	2	10/100	pSOS
JetControl 243	NetArm 40	NetSilicon	33	n.i.	4	10/100	pSOS
JetControl 246	NetArm 40	NetSilicon	33	16	8	10/100	pSOS
JetControl 647	Elan SC520 (5x86)	AMD	133	16	2...512	10/100	?C OS
Inline Controller ILC 200 IB	MC68332	Motorola	16 - 25	128 - 1024(kB)	n.a.	n.a.	VRTXS
FactoryLine Busklemme FL-BK and Gateway FL-GW	NetArm 40-4	NetSilicon	<33	8	n.a.	10/100	pSOS
Remote Field Controller (RFC) 430/450	Pentium	Intel	166	32	8	10/100	VxWorks
RFC 430 ETH/450 ETH	Pentium	Intel	166/ 266	32	8	10/100	VxWorks
Java CMU IAF	Athlon	AMD	700	128	n.a.	2x 100	TimeSys Linux RTOS

TABLE 2. The devices of PABADIS partners - state of the art in industrial devices

In order to establish new solutions in automation like integrated and distributed concepts, a rather broad acceptance is required. This will be achieved, if such new solutions can be applied to legacy devices - the definition of a broad class of targets covering as much as possible legacy devices is the crucial point in this respect. The major aspect in defining this target area is a reasonable compromise between the capabilities of solutions and required resources and platforms - if the reference platform chosen for automation solutions is relatively modest in its requirements, the platform and new solutions will likely gain greater acceptance because legacy devices might fall into the target area. Hence, the platforms used with those legacy devices must be determined in order to facilitate the definition of an accepted reference platform.

In this thesis, the focus is on Java-based solutions, and thus a suitable Java environment will be determined - this leads to the evaluation of Java platforms provided for automation devices and their operating systems. These are summarized in the following table:

OS	Java Platform	Description, Specific Features
?C OS	n.a.	-
VRTXS	n.a.	-
pSOS	Perc VM, Jeode	Perc VM is provided by NewMonics, J2SE 1.3 and J2ME CDC compliant (see [62]) supports the J2ME CDC specification, with Foundation Profile, Personal Profile, Personal Basis Profile (see [63, 64])
VxWorks	Perc VM JWorks	Perc VM is provided by NewMonics, J2SE 1.3 and J2ME CDC compliant • fully compliant with PersonalJava 1.2 specification, based on JDK1.1.8 API, including SDK 1.2.2 security and JNI (see [60, 61]) • J2ME CVM/CDC reference implementation by Sun Microsystems
TimeSys Linux RTOS	JTime	Java 2 ME CVM/CDC 1.0 Platform , RTSJ compliant real time Java facility (see [65])

TABLE 3. Java platforms for the given operating systems

Compared with the Java 2 standard platforms, these platforms can be characterized as limited device platforms so that legacy PLC devices also belong to the category of limited devices (see also section 3.3.2).

Considering these platforms and focusing on a reasonable compromise between the facilities of (possibly plug-and-participate-based) automation solutions and the prerequisites of state-of-the-art automation devices, suggests a reference platform Java 2 Micro Edition with the Consumer Virtual Machine and the Connected Device Configuration (J2ME CVM/CDC; please refer [96]). This platform, however, might also be too resource-consuming for automation devices, and thus a lower dimensioned platform such as the J2ME with the Kilo Virtual Machine and the Connected Limited Device Configuration (KVM/CLDC, refer [92] and [95]), should be chosen as the reference platform in order to ensure the possibility of running automation solutions on as many legacy platforms as possible. **Such a consideration led to the design decision to use the KVM/CLDC as the reference platform for (Java-based) automation solutions.**

Fieldbus Devices

The range of fieldbus devices is rather broad - it covers the entire spectrum of networked automation devices starting at PC-based hardware connected by appropriate adapters and includes controllers and devices such as sensors and actuators in the field. While the first two categories mentioned have already been evaluated, the sensors and actuators remain to be investigated: Sensors usually measure values such as temperature, light, pressure and others, and provide this information to appropriate receivers, e.g., controller devices on the fieldbus. These controllers, in turn, process the values and trigger respective actuators like switches, valves, LEDs, etc.

The range of tasks that sensors and actuators have shows that these devices do not require any further “intelligence” like computational power or memory. Hence, no platform evaluation is necessary.

From this evaluation of automation devices, and a proposal for a reference platform for automation solutions (described in chapter 5, 6, and 7), an excursion to limited devices and their limited device platforms will be made. This further substantiates the arguments that automation devices are more or less limited devices. The Java platforms listed in table 3 are already identified as limited device platforms. Moreover, this limited-device character of targets and platforms is the major constraint for the definition and development of an automation solution as shown in chapter 5, and an appropriate plug-and-participate technology is shown in chapter 6.

3.3.2 Excursion: Limited Devices and Limited Devices Platforms

Automation devices, especially the legacy devices sketched in the previous section, have been characterized as limited devices and devices providing limited device (Java) platforms. However, this was without a formal definition of this device and platform genre. The main reason for this is that there does not exist any real definition - limited devices are generally described by limited resources like computational speed, memory, display size, storage, etc., or characterized regarding their size, or defined by further aspects - for this thesis the limited resources are the fundamental aspect.

Limited resources, however, can usually be measured and explained as discrete values, although a definition based on such fixed attributes is not reasonable: State-of-the-art hardware might have limited device character in a few years in the same way that devices built several years ago are perceived as limited devices today.

In order to get a deeper understanding of limited devices, some examples are given in the following table. It lists everyday's goods - common for everybody:

Device	Features, Prerequisites, Resources
Palm™ Tungsten™ T3 Handheld	• 320x480 Stretch Display • 400MHz Intel® Xscale™ processor • 64MB memory • Built-in Bluetooth • Java™ 2 Micro Edition (J2ME), KVM/CLDC
TG50 CLIÉ™ Handheld	• High-resolution TFT color display (320 x 320 pixels) • High performance processor (200 MHz) • 16 MB RAM, 16 MB ROM • Built-in Bluetooth networking capability • Java 2 Micro Edition KVM/CLDC with MIDP

TABLE 4. Arbitrary Limited devices - everyday's goods (taken from the respective home pages)

Device	Features, Prerequisites, Resources
iPAQ h2215 Pocket PC	• TFT transflective screen with 64,000 colors • 400MHz Intel® XScale™ processor • 64MB SDRAM, 32MB Flash ROM • integrated Bluetooth™ wireless technology • Java 2 Micro Edition CVM/CDC (on Linux)
Siemens SL55 (mobile phone)	• Color screen • 1MB user memory • Java support (Java™ Wireless technology, KVM/CLDC)

TABLE 4. Arbitrary Limited devices - everyday's goods (taken from the respective home pages)

In comparing these listed devices with the automation devices given in table 3, one difference is of particular interest: The automation devices partially provide less resources than these devices (except the mobile phone). Nevertheless, the provided Java platforms are rather similar - usually limited-device Java platforms are offered, at the minimum the Java 2 Micro Edition KVM/CLDC. This platform is a specific limited device compliant (Java) platform providing fewer features than the standard Java platforms, i.e., Java 2 Standard Edition or Java 2 Enterprise Edition. Moreover, the J2ME framework defines extension packages addressing the specific demands of limited devices. Such classes do not belong to the standard classes of Java - the package `javax.microedition.io.*`, for example, provides the `Connector` framework of the KVM/CLDC platform for network access. The relationships between these different Java editions are illustrated in the following figure:

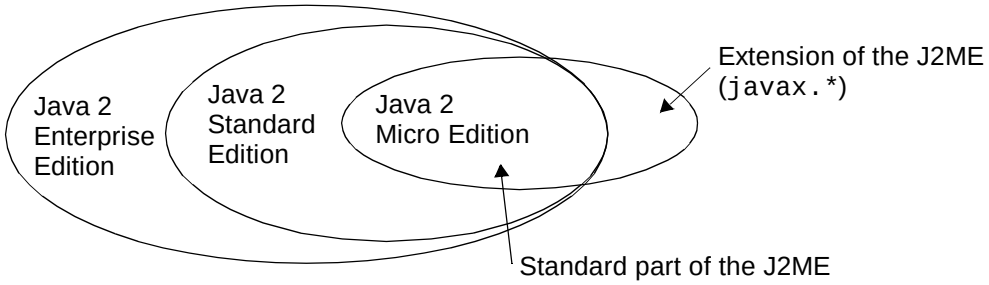


FIGURE 8. The structure of Java platforms

Taking the statements regarding limited devices in general and for automation devices in special into account, the decision to declare the Java 2 Micro Edition KVM/CLDC platform as the reference platform can be justified finally: It tackles legacy automation devices and their platforms and, moreover, it is suitable for the general field of limited devices as sketched in this section.

CHAPTER 4

Intermediate Summary of Prerequisites

This chapter gives an intermediate summary of the prerequisites and, moreover, provides concluding remarks to these facts. It will furthermore show, what has been introduced so far and will bring these facts into an appropriate concept.

The starting point of this thesis was a discussion of the current market situation and the identification of the necessity for suitable algorithms, solutions and technologies that are able to cope with today's turbulent market situations. Automation solutions compliant with such turbulences need to be flexible and adaptable, and thus centralized approaches can be excluded from the discussion - only distributed system-based solutions are applicable to this thesis. The discussion led to the identification of three major facets of this thesis:

- Plug-and-participate technologies in order to provide basic flexibility
- Distributed automation solutions, especially plug-and-participate-based concepts
- Automation devices as target platforms, usually characterized as limited devices

All three facets have been introduced in previous chapters; especially the historical background was sketched, and various technologies and proposals were surveyed. For example, middleware technologies like CORBA, RMI and SOAP were described as the basis for the classical plug-and-participate technologies Jini and UPnP. Both plug-and-participate concepts build the fundament of further discus-

sions in this thesis, and thus they have been evaluated in detail, including an overview of their architecture and their protocols. It finalized in a comparison of their facilities and requirements. From this, their inherent drawbacks were derived. These drawbacks, especially their tremendous resource consumption in terms of memory and network bandwidth, are starting points for the development of a plug-and-participate technology that is suitable for limited device environments. Such a fundamental prerequisite results from the evaluation of state-of-the-art devices and platforms in the field of industrial automation: Devices and their platforms were thus investigated and the results are listed in appropriate tables. The main focus in this investigation was on the resource consumption and on the Java platforms supported by the devices and their OSs. These were finally compared to standard Java platforms, which are the basis for frameworks like Jini. Taking this into account, it can be further explained that Jini, for example, is not well-suited for use in automation environments: Its platform requirements as well as its resource consumption are considerably too high for legacy automation systems and limited devices in general. Moreover, the same arguments apply for UPnP - due to its inherent network load and redundancy caused by its totally decentralized architecture, it is also not well-suited for automation systems.

The third aspect of this thesis aside from plug-and-participate for limited devices is the field of industrial automation systems. In this respect, the general concepts and legacy systems, namely centralized planning systems have been introduced. Also, the trend towards distributed automation systems was sketched, which is motivated by turbulent markets and the lack of flexibility and adaptability of current plant automation solutions. Finally, three representative examples of distributed automation concepts are surveyed: HMS, IDA and PROFInet. The main focus within this evaluation was on the basics and the principles of their architectures. For example, the HMS concept demonstrates initial plug-and-participate approaches, because resource holons freely organize appropriate manufacturing holons. They join and leave such infrastructures as usual in plug-and-participate systems ensuring a certain basic flexibility of HMS. In contrast, PROFInet and IDA could be improved if the current manual assignment of automation objects to underlying physical functions - abstracted by runtime objects - will be automated, e.g., by plug-and-participate concepts. Proposals for such improvements are given in chapter 5 as one basic contribution of the thesis' author in this field. However, these proposals are rather secondary results of chapter 5, which concentrates primarily on a plug-and-participate-based automation solution called PABADIS - Plant Automation Based on Distributed Systems. This solution - innovative in the field of industrial automation - will finally combine all these three facets into an integrated solution. Of course, this thesis does not cover the entire PABADIS approach but instead focuses on one particular and fundamental aspect - a plug-and-participate concept results, which was the main task of the research done at the University of Marburg. It is furthermore the main result of the author's individual research within the project. The PABADIS plug-and-participate concept defines a general framework for the automation field, and is furthermore fundamentally related to the abstract plug-and-participate interface defined in section 2.3.1. A Jini-based implementation of this framework is provided, but due to the inherent requirements of Jini, a more suitable plug-and-participate technology, called Pini, was developed by the author - it is described in detail in chapter 6.

The results of chapters 5 and 6 are evaluated in chapter 7; especially resource consumption measurements will show the applicability of the PABADIS approach on limited devices on the field level. This ultimately approves PABADIS as a solution being compliant for legacy automation systems as well as Pini as an innovative plug-and-participate framework for rather complex applications - even on limited devices.

CHAPTER 5

PABADIS - Plant Automation Based on Distributed Systems

The first part of this chapter integrates all three facets of this thesis, namely plug-and-participate, limited devices, and plant automation. An integrated automation solution results, which is called PABADIS - Plant Automation Based on Distributed Systems. The second section will furthermore tackle related issues of plug-and-participate use on the field level. For example, proposals for enhancements to IDA and PROFInet, distributed control applications and a fieldbus-based plug-and-participate approach relying on appropriate semantic concepts will be investigated. These approaches demonstrate rather basic efforts for plug-and-participate on the field level. However, they are the first efforts in providing flexibility on each plant level, and they point towards integrated solutions that cover the entire automation pyramid.

The PABADIS approach, which is the main focus of this chapter, is situated at the MES level, and thus on top of the aforementioned concepts: First, an overview of the entire approach is given, including the identification and classification of the plug-and-participate concept. Subsequently, this plug-and-participate concept is described in detail, especially the development of a plug-and-participate abstraction for automation and the respective PABADIS implementation of it. The evaluation of the benefits and the already-stated tremendous resource consumption of classical plug-and-participate technologies (given in section 2.3.4) will finally lead to a limited device suitable plug-and-participate technology that is the topic of chapter 6.

5.1 A Paradigm Shift in Plant Automation

Following the arguments given in the motivation, in section 3.1, and also in chapter 4, the turbulences in the markets caused by an increasing customization and individualization trend require a complete customer-orientation of vendors. They have to satisfy strongly individualized customer demands (remember also the Maslow pyramid mentioned in the motivation) in order to survive in the markets. Such an ability to follow new trends, adapt to special customer needs and react quickly to changed environments and demands requires a high degree of flexibility and adaptability of the production environments - plant infrastructures must be re-configurable. This not only affects the field level but likewise the planning level. However, today's rather centralized automation systems lack this re-configurability and flexibility. Distributed systems, in contrast, address such flexibility needs, and thus there is an increasing trend towards distributed automation solutions (see also [50] as well as chapters 3 and 4). This trend not only signifies a change from traditionally centralized automation and planning to distributed solutions that are more suitable to today's changing markets - it defines a paradigm shift. This paradigm shift to distributed systems is accompanied by a revolution in the main facets of plant automation - namely hardware, software, and communication.

This revolution is strongly related to the trend of providing integrated solutions that seamlessly cover all levels of plant automation. For example, a rather heterogeneous field of hardware is common with specific solutions and technologies at each level - hence, the provision of a certain homogeneity constitutes a major improvement. This current heterogeneity furthermore impacts the software field, which must cope with the wide variety of platforms - a platform-independent programming language increases the efficiency in this field (see also [4]). Both together can be further improved by modularity, which increases the possibility to re-use already existing and well-proven components. Finally, a standardized communication over all levels ensures the intended interoperability.

A two-dimensional matrix is covered by this paradigm shift. This matrix is drawn by the dimension of the automation components hardware, software and communication, and the second dimension given by the levels of the automation pyramid, namely ERP, MES and machine level. The resulting nine fields of that matrix comprise specific state-of-the-art solutions, which are based upon respective standards common in that field (see also figure 9). In particular, such strongly diversified fields must be harmonized as one objective of this paradigm shift. It is a motivational point for the PABADIS solution and defines secondary goals for PABADIS, while the main goal of PABADIS concerns the transition from centralized systems to distributed automation.

5.2 Objectives and Overview of the PABADIS Solution

PABADIS is a research project funded by the European Commission under IST-1999-60016. It is composed of 10 partners from industry and academia (4 universities/research institutes and 6 companies)

in Austria, France, Greece and Germany. Moreover, under the IMS program of the European Commission institutions from Switzerland, Canada and the US also participated in the project.

PABADIS furthermore stands for Plant Automation BAsed on DIstributed Systems and embodies a distributed automation solution that follows the paradigm shift in order to cope with the turbulences in the markets. Following this new paradigm, PABADIS intends to simplify the strongly diversified fields drawn by the aforementioned matrix. This results in a revised matrix, where each automation component is based upon a common platform as illustrated in the following figure:

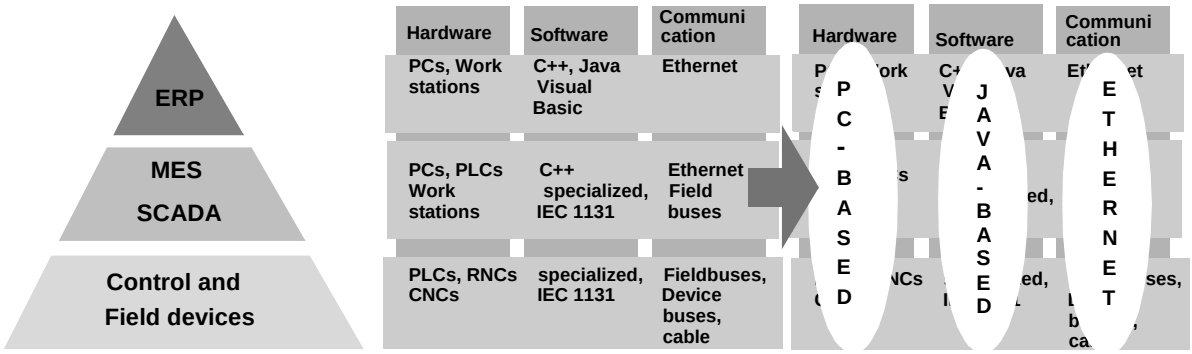


FIGURE 9. The Automation Pyramid and the automation facets illustrating the change to homogenous solutions

The use of Java and Ethernet in the software and communication fields is one of the main goals of PABADIS. Such common platforms allow a seamless integration of all levels of the automation pyramid into an appropriately integrated automation solution from the ERP level down to the machine level (see [68], chapter 1). This ensures a vertical integration.

The second aspect to be covered is the horizontal dimension as well as the horizontal integration throughout all automation facets. For PABADIS, especially the field level was of interest - the major goal in this respect is the provision of a homogeneous view and simultaneously a standardized access path to the basic functionalities. This requires a dedicated abstraction of underlying functionalities, because entities resulting from the vertical integration are generally interested in functions rather than in particular hardware. PABADIS therefore provides this kind of horizontal integration via plug-and-participate technologies (see [74] and [76], chapter 1) defining the required abstraction. This furthermore ensures a high degree of flexibility.

These aspects can be aggregated to the main objectives of PABADIS:

> *PABADIS mainly aims at providing an **integrated automation solution** covering the entire automation pyramid in order to allow a **high degree of flexibility and adaptability** of automation systems, depending on the market situation on one hand, and of system-related issues like system load, etc., on the other. In particular, this solution is based upon a **distribution of MES functionalities and tasks** as much as possible. The integration is mainly based on **common platforms for each particular automation component** hardware, software and communication. These platforms cover all levels of the automation pyramid as shown in figure 9.*

Well-known and established technologies like Java, Ethernet and PC-based hardware will be therefore the basic platforms of PABADIS.

The solution results in a seamless integration of all automation levels into the information and process flow as well as in the consideration of all automation components based on technologies and platforms spanning all these levels as depicted in figure 9: The horizontal integration is realized by plug-and-participate technologies, while the vertical integration is strongly related to (intelligent and cooperative) mobile and stationary software agents (see [66, 68]).

5.2.1 The PABADIS Concept and Its Components

Remembering the PABADIS objectives to provide an integrated solution covering all levels of the automation pyramid and thus all relevant plant levels, its basic idea follows the paradigm shift by providing a decentralized functionality - PABADIS mainly focuses on the replacement of traditionally centralized MES systems with decentralized mechanisms based on distributed system facilities. In particular, dedicated MES functionalities will be implemented in mobile and stationary software agents. This distribution considers the reasonability of a decentralization of such facilities (see also [66]). It is the main fundament of the PABADIS concept, and delegates the complete control of product processing to agents, which in turn have to navigate their work pieces through the plant infrastructure. This navigation includes scheduling of tasks, and the determination of suitable machines/functions for task processing and related resource allocations. These sketched facilities require a constantly up-to-date view of the underlying function infrastructure in order to cope with changes and adaptations in the environment. If furthermore a homogeneous view can be ensured, it allows on-demand decisions about what function to be used for a defined task - the horizontal integration provides such a required abstraction and flexible infrastructure based upon plug-and-participate technologies.

In concluding this first overview of the PABADIS concept, an adaptation of the legacy automation pyramid becomes necessary: The traditionally monolithic MES level is replaced by autonomous mobile and stationary software agents illustrated in the following figure by loosely-organized filled cycles. Also the monolithically drawn field level is separated into blocks demonstrating the definition of independent homogeneous modules representing abstracted machine functions. The integration of all levels is

finally realized by homogeneous interfaces, which are shown by the corresponding connection points directing to the ERP level and to the field level:

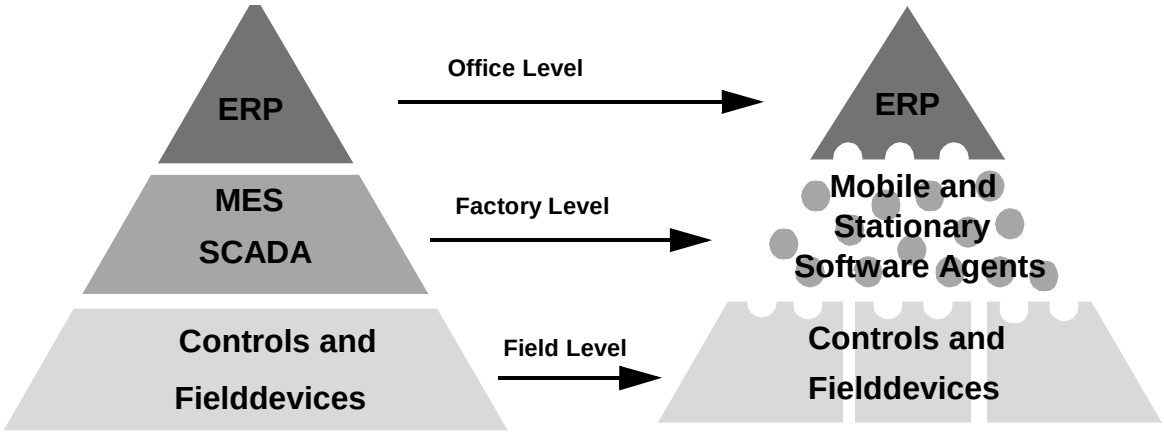


FIGURE 10. The CIM Automation Pyramid adapted to the PABADIS concept

Such interfaces ensuring this integration are represented by the so-called PABADIS-Agency, which connects the PABADIS automation solution to arbitrary ERP systems, and Residential Agents (RA), which provide generic access paths to machines as well as to the PABADIS-Agency. These RAs furthermore ensure the standardized and homogeneous communication between all levels. This homogeneity in communication is achieved by agent communication based interaction patterns. The main focus of RAs, however, is on the representation of PABADIS entities in the community such as machines, functions or even the PABADIS-Agency. Residential Agents are the main component of so-called Cooperative Manufacturing Units (CMU), which abstract the underlying machines by providing their functions in a homogeneous fashion. These functions are offered to the PABADIS system via plug-and-participate facilities. They are moreover described in standardized form via so-called XML-formatted Capability Descriptions (CD). The counterpart of such CDs are the likewise XML-formatted Work Orders (WO) that are generated by the PABADIS-Agency, which extracts such orders from more general Manufacturing Orders (MO) issued by a connected ERP system (see especially the deliverable [71] for the structure of orders and CDs).

Remark: A Manufacturing Order describes an entire product such as a car, while Work Orders describe the individual parts of Manufacturing Orders. For example, a possible Work Order could be for the left front door of a car described in a Manufacturing Order.

Each Work Order is finally assigned to a dedicated mobile Product Agent (PA), which navigates its order through the CMU infrastructure in order to accomplish the specified tasks.

In addition to these already depicted agents - RAs for function provision and PAs for product processing - PABADIS defines a third category of agents that deal with production-, but not product-related tasks, so-called Plant Management Agents (PMA). Agents of this type can be both mobile as well as stationary, depending on their specific tasks. An example of such an agent is the SCADA PMA, which is responsible for setting up SCADA relationships between a SCADA CMU and arbitrary CMUs that want to be supervised by SCADA facilities. Tooling PMAs, supporting the PABADIS system with tool-

ing facilities, are a second example. Tooling is usually not concerned with a certain product/production step, but of course with the production itself. For more details on agents, please refer to the agent-related deliverables of PABADIS, namely [66, 69, 70].

The fourth component of PABADIS is a Lookup Service (LUS) providing entities with necessary information about the infrastructure, e.g., an overview of functions in the system. However, the PABADIS concept relies on the provided functionality. This means, the PABADIS approach requires a facility, which is able to provide the required infrastructure overview - its realization, in turn, essentially relies on capabilities of the underlying plug-and-participate system, as it will be shown in section 5.3.3.

A recapitulation of the PABADIS components and a first impression of the entire system is illustrated by the following figure, which furthermore shows basic relationships between the components:

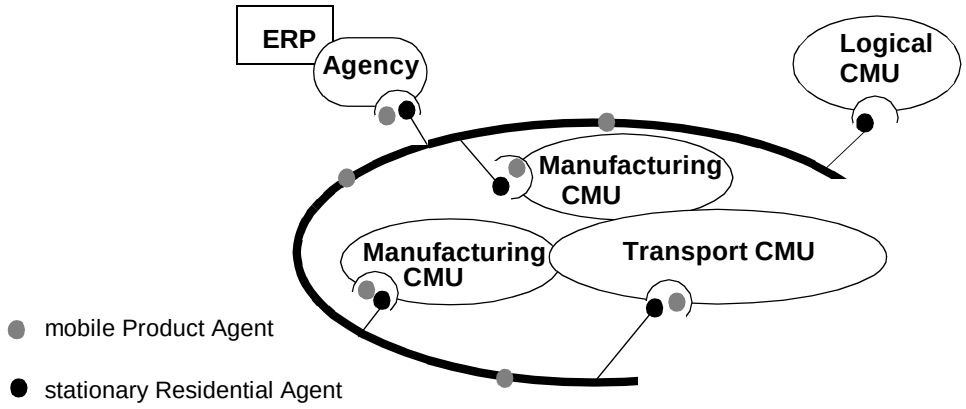


FIGURE 11. General topology of a PABADIS plant

The figure especially demonstrates the basic PA behavior: PAs migrate to CMUs in order to process their tasks. They first have to request the LUS component for a CMU, which is able to perform the required task. After deciding which particular CMU it will use, the PA migrates there and starts the task processing. Such a fundamental PA behavior implies that a PA defines its path through the plant infrastructure independently and on its own - a complete decentralized planning facility results. It thereby fundamentally relies on the plug-and-participate system (see also [85]) with regard to the retrieval of CMU references. Moreover, a complete synchronization between work pieces and PAs as control instances is achieved - PAs are always with their work pieces, and thus they control the product processing locally. This means, no remote control is necessary, and network failures usually cannot harm the processing control. Such facilities, namely the completely decentralized planning, the synchronization of PAs and their work pieces and finally the local control of function access are the fundamental innovations of PABADIS.

This overview of the PABADIS concept, as well as the introduction of its basic components has also sketched the part of the project that was evaluated and developed at the University of Marburg: the provision of a generic machine platform and a generic plug-and-participate concept. Both topics are the main contribution of the author to the project, and this thesis will document the achieved results.

This documentation furthermore comprises aspects that were developed in cooperation with other partners from the project - a strict separation of the author's individual contribution and results developed in cooperation is rather difficult, or even impossible due to the complexity of the defined solutions and dependencies between different parts of the PABADIS concept: The main objective of my work within PABADIS was basically concentrated on the development of the CMU architecture. The essential parts in this are the definition of cooperation pattern for these CMUs, and their transparent representation in the PABADIS infrastructure - three particular aspects have been distinguished: First, the plug-and-participate abstraction and the general behavior patterns of CMUs within the community. This also has an impact on agent behavior, especially on RAs, which has also been covered during this work. A second major aspect was the provision of special CMUs and facilities that provide dedicated plug-and-participate-based mechanisms. Both, the plug-and-participate concept as well as the dedicated plug-and-participate based CMUs/facilities are innovations in the field of industrial automation - although, e.g., HMS already has shown first plug-and-participate aspects, PABADIS is the first concept fundamentally relying on this technology genre.

The third and most fundamental part for the work for this thesis was the development of a plug-and-participate technology for legacy automation devices - Pini. This technology is intended as a testbed for the PABADIS solution, especially on limited devices and limited device platforms. Pini is described in more detail in chapter 6, while a Pini-based CMU is sketched in chapter 7.

Prior to a detailed description of the topics classified above, the next section will describe the PABADIS system and its general behavior. This especially comprises an illustration of the flow of an order through a PABADIS system.

5.2.2 General Behavior Study of a PABADIS System

The basic behavior of a PABADIS system covers two aspects, which are in direct correlation to both integration directions: First, the creation of the underlying system infrastructure is described comprising the setup of the function network and creating the connection to the ERP level. Second, the agents, namely the PAs, are covered by showing the life cycle of a PA. Agents, especially PAs, realize the vertical integration of all levels. Moreover, the PA life cycle also illustrates the interlocking of both integration directions as well as the relations between PABADIS components.

Setup of Function Infrastructure

This refers to the use of plug-and-participate facilities in order to offer the CMU functionality to the PABADIS community. It mainly reflects the upload of the relevant parts from the CMU's Capability Descriptions. Likewise, specific (maintenance) operations are performed in order to ensure a robust and up-to-date infrastructure. A second aspect of this process is the connection of the PABADIS-related CMU modules to the underlying physical functions. However, these specific operations are transparent for the operators and are defined by the PABADIS concept.

Connection to the ERP System

Turning the focus over to the upper levels, an ERP system needs to exist so that appropriate customer orders can be evaluated to corresponding Manufacturing Orders. These XML-formatted MOs are the basis for the Work Orders of PAs. In particular, the PABADIS-Agency obtains the Manufacturing Orders from the ERP system and decomposes them into several Work Orders. This decomposition

basically utilizes the tree-structure of Manufacturing Orders, since a Work Order results from an In-order traversing of the MO tree, as illustrated in the following figure:

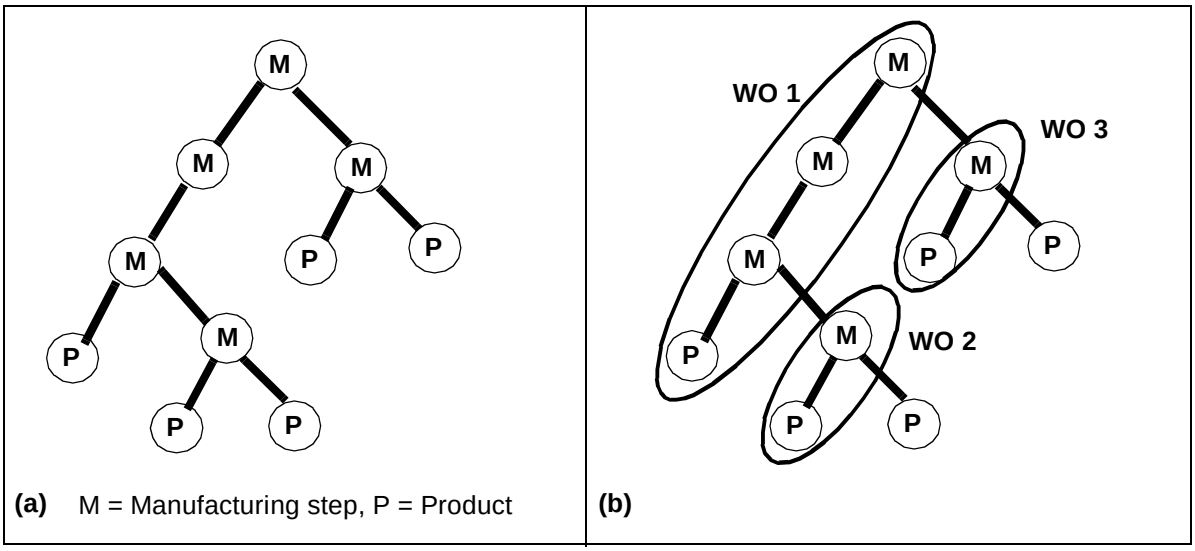


FIGURE 12. Manufacturing Order hierarchy and the corresponding Work Orders

Moreover, this is also related to the underlying infrastructure, especially to the available functions. For example, processes described in the Manufacturing Order must be assigned to specific functions provided by the CMU network. This mapping therefore requires an overview of available functions in order to ensure that a created Work Order can be processed properly (please refer also to [72] for more information regarding the PABADIS-Agency, and section 5.3.4, especially the subsection on the Plant Structure Observer)

Product Agent Life Cycle

After Work Orders have been created with respect to available functions, they are assigned to PAs, which in turn are initialized - for each Work Order, exactly one PA is launched and its corresponding

Work Order is assigned. This creation of a PA and the subsequent upload of its associated Work Order is the starting point of the PA's life cycle:

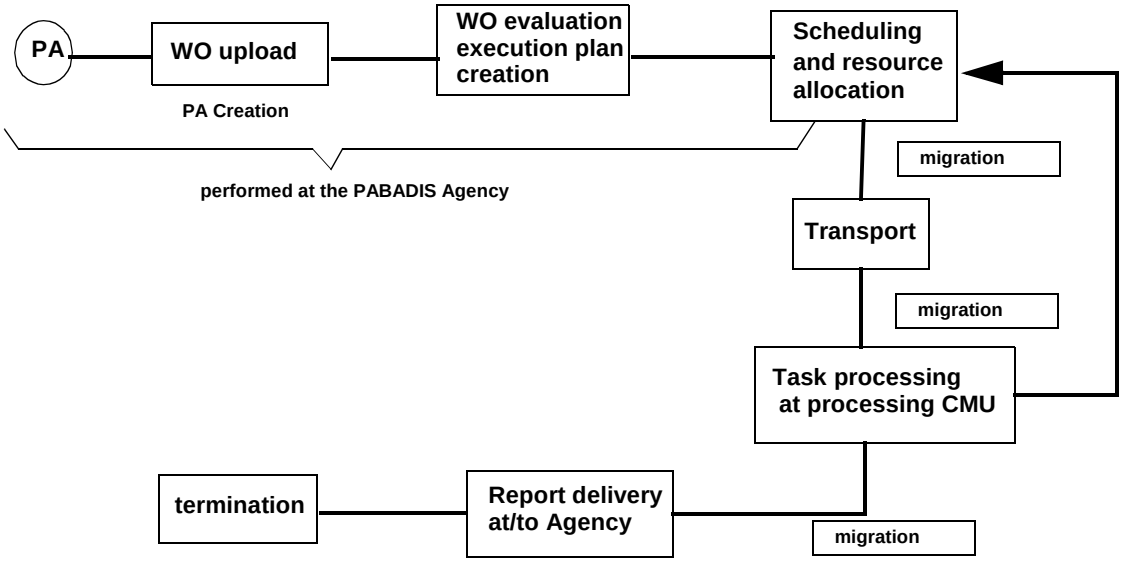


FIGURE 13. The basic life cycle of a PA

After a PA has received all required information, it starts to evaluate its Work Order and a so-called execution plan is defined. It basically lists all possible sequences of tasks, depending on available functions in the underlying PABADIS infrastructure and the current topology defined by the transport system. Out of these paths, the PA chooses the most efficient path (with respect to certain criteria). In the next step, the PA starts to schedule the tasks of the chosen path: It requests the LUS, and thus the underlying plug-and-participate system for suitable CMUs, and finally migrates to the chosen CMU. Of course, the entire procedure is more complex than illustrated here - it furthermore may contain transport facilities, synchronization processes and operations, a lookahead ensuring that the PA does not run into a dead end, etc. However, such details are not necessary for this overview, which only intends to demonstrate the general behavior.

When the PA is finished with its Work Order - whether it was completed successful, it is recalled/ stopped by the PABADIS-Agency for whatever reason or the PA simply detects that it cannot finish its order - the PA returns to the PABADIS-Agency, leaves its reports there and finally terminates.

The general behavior of a PABADIS system is only one aspect of this section, since it likewise covered the particular CMU behavior, especially the dedicated plug-and-participate behavior patterns. Moreover, also the plug-and-participate related agent behavior was briefly sketched. Details regarding the plug-and-participate-related concept of PABADIS are defined in full in the following section.

5.3 Plug-and-Participate in PABADIS

The fundamental focus of this section lies on the PABADIS plug-and-participate concept resulting from a general automation-related plug-and-participate abstraction. This abstraction is closely related to the abstract plug-and-participate interface given in section 2.3.1. Interlocked with this description of the automation abstraction layer is a Jini-based reference implementation showing under-the-hood details of the PABADIS plug-and-participate solution. The independency of the general automation-related plug-and-participate abstraction as well as the independency of the PABADIS solution of specific plug-and-participate technologies is illustrated by a description of a second reference implementation based on UPnP. While the Jini-based reference implementation is simultaneously the PABADIS reference implementation, the described UPnP-based concept is of hypothetical nature - it simply demonstrates how to realize PABADIS using UPnP. These three facilities, the abstract automation-related plug-and-participate interface as well as the mentioned two reference implementations basically result from the editor's work within the project. Moreover, these contributions are the motivation for a further result: For measuring the resource consumption of PABADIS CMUs consequently only the Jini-based implementation can be used. A comparison of such measurements with given platforms and prerequisites of state-of-the-art automation devices results in an obvious gap between requirements and prerequisites: The reference implementation cannot be applied to legacy devices, and thus a suitable technology such as Pini is required - this will be described in chapter 6.

However, such investigations require detailed knowledge of the internal CMU structure and the machine platform. This internal structure and the main CMU components are sketched in an overview fashion in the following sections. The knowledge of the CMU architecture allows a classification of the plug-and-participate module to be developed. Moreover, the design decision for plug-and-participate technologies must be justified, which proves the applicability of this technology genre in automation, and especially in PABADIS. The necessity of plug-and-participate has already been shown in the motivation and chapter 3.

5.3.1 Design Decision for Plug-and-Participate Technologies

PABADIS is based on two main technologies, which ensure the required vertical and the horizontal integration. In particular, agent technologies are used for the vertical integration, while the horizontal integration relies on a plug-and-participate concept. Both are dedicated technologies for distributed systems, but have rather different objectives. For PABADIS, however, specific facilities are required that could be provided by both genres simultaneously. Such similarities are only with respect to the required features and facilities defined in both technologies - as described in table 5.

Remark: The listed agent features result from the investigation of the PABADIS requirements. It is furthermore checked, whether the technologies provide the features, while a detailed description of the basic agent system facilities is not necessary. For interested readers, such features are concluded in section 7.3 and [104].

The table therefore compares agent systems and plug-and-participate facilities with respect to the general PABADIS structure and behavior of its components:

Feature	Agent Technologies	Plug-and-Participate Technologies
provision of …	agent related features	service - service user infrastructures
spontaneous networking	yes, naturally provided	yes, naturally provided
lookup feature	yes, tailored to agent search	naturally provided, tailored to service search
passive information system	partially provided, but often broadcast based	naturally provided
self-maintenance	often not supported	naturally supported
communicative	implementation specific, but normally supported by agent systems	implementation specific

TABLE 5. Comparison of agent and plug-and-participate technologies

Obviously, the features are, abstractly seen, very similar. Relative to the investigated topics, both technologies provide similar behavior patterns such as spontaneous networking and lookup facilities. However, spontaneous networking in terms of agent systems refers to the creation of agent communities, while the plug-and-participate focus mainly relies on service - service user infrastructures. Also the lookup facilities differ, although their general equivalent meaning: Plug-and-participate systems provide requesting entities with references to dedicated services (see, for example, Jini's proxy concept in conjunction with the lookup protocol, or UPnP's advertisement facility), while agent systems rely on the provision of agent references. Such a general agent system behavior seems to be best fitting to the PABADIS concept, because CMUs are represented by their Residential Agents, and thus a lookup request provides an address for the agent. However, PABADIS agents request the system for functionality rather than for specific agents. This, in turn, perfectly fits to plug-and-participate concepts. A PABADIS CMU registers all its functions separately as a single service, and a RA-reference returned simply offers the standardized access path to the dedicated function. This leads to the provision of references of services rather than references of agents. Such concept obviously fits best to plug-and-participate facilities rather than to agent systems - plug-and-participate aims at providing service - service user relationships, while agent systems provide a search facility essentially tailored to agent search.

Further important aspects are the provision of passive information systems and certain self-maintenance facilities - plug-and-participate systems naturally provide such fundamental features, while agent technologies do not necessarily offer them.

Considering these fundamental differences, although the abstract similarities of both technologies, this led to the design decision within the PABADIS consortium to use plug-and-participate technologies for infrastructure provision instead of customized agent systems. This decision was formed with the expertises of the involved partners, especially from Vienna and Magdeburg for agent-related input, and from the thesis' author for the plug-and-participate related facilities. During this evalution, the required features were listed, and possible solutions were discussed appropriately using both technologies. For agent systems usually specific customization efforts became necessary - customizing agent systems led to non-standard solutions, while plug-and-participate technologies are generally sufficient - a plug-and-participate concept finally fits better to automation needs.

Such automation needs and specific PABADIS requirements for the CMU infrastructure lead to a general and generic CMU architecture that mainly controls the general behavior of CMUs. The architecture is depicted in the following section, together with general CMU behavior patterns also classifying the required plug-and-participate facilities within the CMU infrastructure.

5.3.2 General CMU Structure and Behavior Patterns

The second prerequisite for the plug-and-participate concept aside from the sketched design decision is the evaluation of the general CMU structure and its basic behavior. It is mainly related to the CMU components and modules that are modeled on its architecture. Likewise the basic processes within a CMU are shown in this section. This partially details the PABADIS concept and illustrates the horizontal integration, which is achieved using a homogeneous view of all functions of all CMUs. This view, in turn, results from the abstraction of functions and from dedicated CMU components: A CMU basically consists of four main components (controlled by a Residential Agent as one of the components):

- **Residential Agent (RA)**
  The RA represents the CMU and its functions in the PABADIS system - it is the head of each CMU and oversees its particular local system behavior. This control ranges from triggering almost all important CMU activities (like the community join) to coordinating/organizing the local system appropriately. Such coordination and organization efforts also reflect the provision of access to underlying functions as well as further CMU components and resources. Put more abstractly, the RA offers the CMU interface to the PABADIS community, and thus ensures a standardized communication closely related to agent communication.
- **Residential Agent to Function Interface (RAFI)**
  This component separates the PABADIS logic from underlying physical automation functions - it abstracts the automation functions and provides an appropriate generic access path. This abstraction realizes the homogeneous view of the agent system to automation functions by an abstract finite state machine called the Function Control Module (FCM), which models the general behavior of automation functions. The FCM is the fundamental component ensuring that each PA can access each function in always the same fashion. It furthermore loosely couples the non-realtime constrained PABADIS logic and the automation environment with its hard realtime constraints (see [25] for the FCM). Of course, such generic access also requires CMU- and function-specific data, which are provided by the Capability Description defining parameter names, parameter types, function names, vendor identification, process related data, etc. From this information, the system-relevant data are extracted and published to the PABADIS community via plug-and-participate facilities (general information regarding the CD and the RAFI can be found in [25] and [52]).
- **Common Feature Module (CFM)**
  This module contains all mandatory facilities. They are identical for each CMU, such as the plug-and-participate module, the XML component, the MES tools module, the HTTP server, the data and memory manager, the SCADA component, etc. (see also [67] and [74]).
- **Automation Function (AF)**
  This component represents the real manufacturing capabilities of a CMU, e.g., drilling, welding, etc.

All these components must be initialized during the startup process of the CMU: The RA is started first, because it is the control center of a CMU. Subsequently, all underlying components are launched after

the evaluation of configuration data, such as a special LUS URL, CD path, etc. from a CMU specific configuration file.

The start up process in detail mainly concerns the RAFI and CFM initialization. Because they require information from each other, both components start in a multi-stage process: For example, the CFM needs access to the CD, which in turn requires an XML parser that is provided by the CFM. However, this interlocking will not be explained in detail here, but the start up of the components in general will be briefly sketched:

The RAFI start mainly concerns the initialization of the FCM and the accompanied connection to the underlying automation functions. As shown in [25], the FCM loosely couples the PABADIS logic and the vendor-specific, often hard real-time-constrained automation part. Herein different PABADIS specific connection types can be determined, leading to different CMU types:

- COM-linked CMU: Comprises a complete separation of the PABADIS logic and the automation function via an arbitrary communication system. Both parts can run on different hosts/hardware.
- OS-linked CMU: The automation function and the PABADIS features are separated by a communication connection via the operating system - both parts usually run on the same host.
- Java-linked CMU: In this case the PABADIS logic is connected to the automation function through the Java Virtual Machine. Such a CMU provides Java-based functions running in the same JVM as the PABADIS control logic - the control code is arbitrary Java code.

This FCM concept belonging to the generic machine platform obviously ensures a high degree of abstraction, and hides any details regarding functions access from agents. It is the fundament for providing functions via plug-and-participate to the infrastructure ensuring a generic access. In [25] - published by the mainly involved colleagues and the thesis' author - detailed information regarding this important PABADIS facet, as well as in PABADIS deliverable 3.3 ([74]) a description can be found.

The concepts described in this deliverable are the results from a PABADIS task, where the author took the leadership - the resulting CMU infrastructure is illustrated in the following figure:

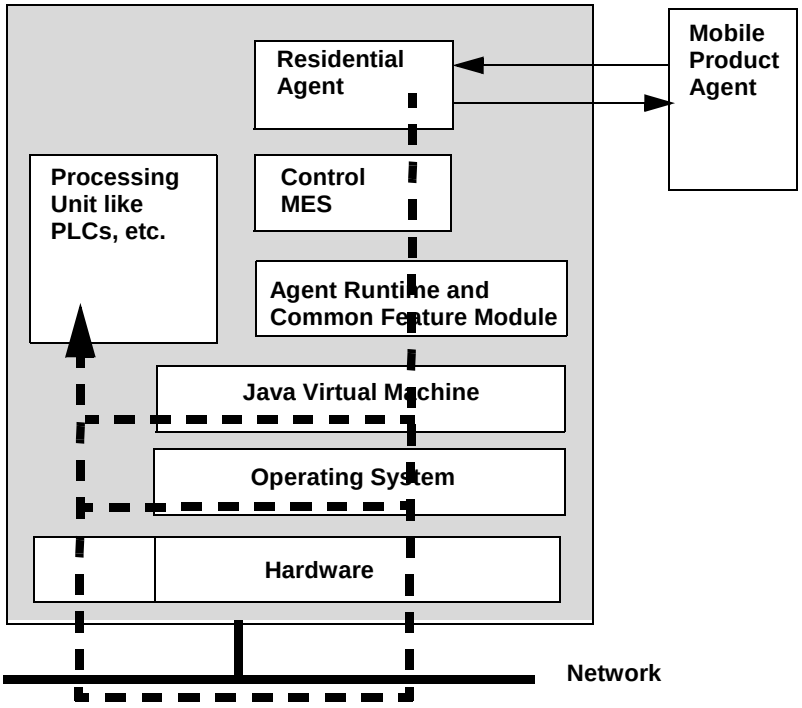


FIGURE 14. The CMU types with respect to the separation of automation functionality and PABADIS logic

The second main part of the CMU is the Common Feature Module, which needs to be initialized properly. For example, it creates basic structures that support the RA in controlling the CMU behavior, such as the schedule mechanisms, HTTP server, tracer module, and finally the plug-and-participate module. While the first three mentioned components mainly support CMU related processes, the plug-and-participate module offers the CMU functionality to the PABADIS community. The following section therefore concentrates on this facility: Its initialization involves the evaluation of the provided functions, the preparation of relevant data and finally the announcement of them. This announcement describes the join to the community as defined in the abstract plug-and-participate interface. Accompanied by such registrations are appropriate maintenance operations in order to keep the community as robust, stable and reliable as possible.

In addition to these rather function provision related aspects, the plug-and-participate module also offers facilities for function search as well as mechanisms for participating in the passive information system.

The last aspect to be sketched is the behavior of the CMU in the community: Usually the CMU behaves passively and waits until it is contacted by an external entity. For example, a PA might migrate to the CMU and then the PA passes its task to the RA. The RA, in turn, triggers the FCM so

that the task of the PA is processed properly. A RA furthermore waits for the incoming scheduling request of PAs - they usually negotiate with the RAs for free slots and suitable functions.

This description of the CMU architecture and its basic behavior also covered the plug-and-participate facility of CMUs - it is contained in the CFM of each CMU. From this general plug-and-participate behavior, an abstract automation-related plug-and-participate interface will be derived in the following section, which furthermore comprises the documentation of Jini- and UPnP-based reference implementations of this interface.

5.3.3 Automation-Related Plug-and-Participate Abstraction and its PABADIS Reference Implementation

Of the three main facets of this thesis, this sections concentrates on two of them, namely plug-and-participate and plant automation - the main goal is to integrate them to an automation solution based on plug-and-participate concepts. This integration will be shown in two parts: A general plug-and-participate abstraction for automation derived from appropriate requirements in the automation field, and an implementation of this abstraction. The implementation-related part is furthermore divided into a Jini-based implementation and a hypothetical implementation based on UPnP. Both aspects, the abstraction as well as the reference implementations, result from the author's research efforts in the third PABADIS work package and concerned tasks that were in his field of responsibility as work package as well as task leader.

5.3.3.1 Requirements for Plug-and-Participate in Plant Automation

As already shown in the motivation, the reasons for plug-and-participate in plant automation are rather manifold. Within this section, the essential requirements will be listed and evaluated in order to derive a dedicated abstraction for automation in general.

The customer orientation and the resulting necessity to satisfy the customer demands require flexibility and adaptability of vendor capabilities. They must be able to react to trends efficiently. This especially covers two facets of the automation field - the field level and the planning system on top of the field level. Both are rather interlocked: If the infrastructure changes, the planning system must adopt these changes, and vice versa - if the planning system decides to adapt the machine level infrastructure to the current situation, the field level in turn must be able to react appropriately to such required changes. This means, the planning system needs an up-to-date view of the underlying infrastructure, while the field level must allow entities to join and leave the community without major interventions or configuration efforts. Such requirements result in ad-hoc networking facilities of the automation field level, which provides mechanisms for functions to spontaneously join and leave the community.

A further aspect concerns the reliability of such infrastructures - spontaneously organized entities are usually independent of each other and thus require dedicated mechanisms in order to keep such communities as up-to-date as possible. This mainly affects the up-to-dateness of the planning systems, which need information as to whether all known entities are still alive. Such up-to-dateness of the planning system moreover requires mechanisms allowing the retrieval of information about available functions, either in active or passive fashion. While the active information retrieval basically means that entities actively request the community for information, the entities of the passive information system receive information about changed infrastructures. This, however, likewise requires appropriate active mechanisms within the community such as management facilities, tracking mechanisms, etc.

Considering all these facilities with respect to the situation of today's markets, this will lead to a high degree of flexibility of both the planning system and the function infrastructure, which finally leads to increased efficiency of the entire plant automation field, and thus to a next generation automation paradigm - in other words, a paradigm shift.

5.3.3.2 An Abstract Plug-and-Participate Interface for Plant Automation

Summarizing the requirements collected in the previous section, an interface for a general plug-and-participate concept in the field of plant automation can be derived. These requirements will be recalled in the following list, which is the basis for the ultimate aggregation into a Java interface - as an innovation in the automation field resulting from the individual research done by the author:

- *alleviated configuration* of function networks (integration of new functions, removal, and change of entities)
- *self-maintenance* of the system, and thus a certain degree of reliability and robustness is required
- facility to *search* for required *functionalities in active fashion*
- *(passive) information system* allowing for an always up-to-date view of the infrastructure.

Although these requirements are aggregated into a Java interface (as shown in figure 15) they are completely independent of any particular plug-and-participate technology and programming language. This Java interface defines the final abstraction of the determined requirements:

```
public interface IPnPModule{
(1)   public XMLObject[] joinCommunity(String raName, String address,
                                       boolean flag);
(2)   public boolean adaptLeaseDuration(long newDuration);
(3)   public boolean changeAttributes(int id, XMLObject template,
                                      XMLObject newValues);
(4)   boolean addAttribute(int id, XMLObject newAttribute);
(5)   boolean removeAttributes(int id, XMLObject template);
(6)   public Matches[] lookup(SearchPattern pattern, int maxMatch);
(7)   public long remoteEventNotifier(int transition,
                    SearchPattern pattern, EventNotifier notifier);
      public void removeRegistration(long id);
      public Vector getFunctionDescription();
      public String getFunctionID(FunctionPnPID pnpid);
      public FunctionPnPID getFunctionPnPID(String functionName);
      public boolean terminate();
}
```

FIGURE 15. The automation-related plug-and-participate interface

This interface is the basis for the following plug-and-participate reference implementation as part of the PABADIS framework, which is based upon the Jini network technology. Subsequent to this Jini-based implementation, a hypothetical implementation of this interface based on UPnP is described. This especially illustrates the independency of the interface of specific plug-and-participate technologies, and in this respect also the independency of the PABADIS concept of specific technologies.

5.3.3.3 A Jini-based Reference Implementation for PABADIS

The Jini-based reference implementation of the automation plug-and-participate interface is contained in a dedicated module comprising all facilities. Further CMU components always access this module through the interface, and thus the implementation is completely transparent for them and for other PABADIS entities. Although the module generally acts autonomously, it requires references to certain CMU components, such as the RA, in order to inform about specific events in the community and in the module itself. Furthermore, a reference to the RAFI is required, especially to the CD, in order to extract relevant information of the CMU. These relationships are established during the initialization - the components simply exchange their references.

After the initialization phase, the RA can trigger several actions in the plug-and-participate module, such as the join process, attribute handling, participation in the passive information system, searching for functions or adapting the lease duration. All of these facilities and their accomanied operations will be described in separate subsections.

5.3.3.3.1 Community Join

The community join process is one of the most fundamental aspects of the plug-and-participate module. It allows entities to transparently participate in the community, and specifically to provide their facilities and functions to other entities. The process itself is triggered by the RA of the CMU, which has to provide specific information for the join process:

- RA name
  This name is offered to the plug-and-participate system within the Jini proxy of the particular functions. With the help of this name, PABADIS entities establish the connection to the CMU - they connect the RA in order to access the required CMU functionality.
- a LUS address
  This LUS address specifies a dedicated LUS, with which the functions of this CMU must be registered. This ensures that the CMU functions are registered with at least the specified LUS, if it is active. If no specific LUS name is given, the functions will be registered with all LUSs in the community, once they are discovered.
- a boolean flag
  This flag indicates, whether the functions are allowed to be registered with LUSs other than the one specified. If there is no LUS address specified, the flag is always considered to be "true". Otherwise, if it is set to "false" and a LUS name is specified, the functions are only allowed to be registered with that particular LUS.

The information is passed to the plug-and-participate module by invoking the `joinCommunity(..)` method (see the interface in figure 15, line (1)), which also starts the join procedure. It describes a multi-stage process: first, the evaluation of the aforementioned information; second, the preparation of data to be announced to the community; and third, the registration process and the accompanied eval-

uation of the registration acknowledgement data. The evaluation of input data of the procedure is not of further interest - the data simply cause the join process to use different subroutines, but the general behavior is always the same, namely the registration data preparation and the registration itself.

Registration Data Preparation

With respect to Jini's registration facility three particular parameters are required:

- A `ServiceID`, if one is already assigned - in PABADIS, this is usually not the case.
- Provision of the service proxy. In PABADIS, this service proxy carries the address of the local RA, so that entities can use this address in order to connect to the CMU. The proxy simply stores this address as a `String`, so this way remote class loading can be avoided and no resources are wasted - for each function, the same proxy class is used regardless of the particular features of the functions.
- The attribute sets, where the essential CMU information is stored. These attributes are extracted from the CD and are furthermore extended by additional information. Moreover, in order to avoid class loading, to ensure a generic data structure and finally to alleviate the obligatory matching, these attributes are stored in special objects in a dedicated XML format - they are called tag-based trees and are implemented in specific Java `XMLObject` instances (tag-based trees are described in [24]).

This attribute preparation is also a multi-stage process comprising the evaluation of function data, SCADA data, CMU identification data and finally the retrieval of the HTTP port of the CMU's HTTP server.

Function data are extracted from the CD - they are indicated as plug-and-participate relevant in the CD. A request to the XML parser is made, resulting in a set of XML-structured data describing the particular functions. The structure is as follows:

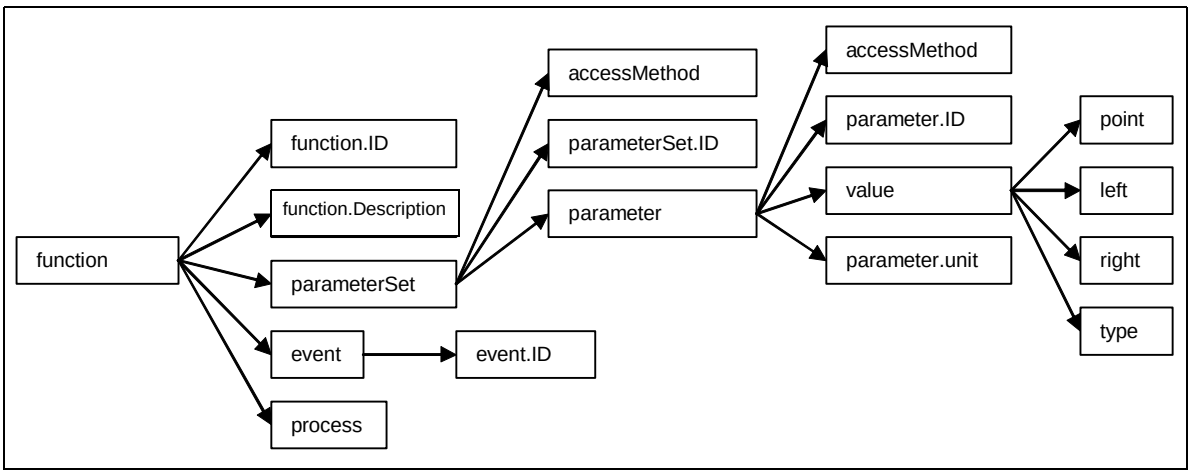


FIGURE 16. The general structure of the data sent to the LUS during the registration process

In the same fashion, the SCADA data are retrieved from the CD via a dedicated XML parser request. The return value from the parser indicates whether this CMU must be supervised by a SCADA system. It is therefore sufficient to aggregate this information into a special attribute instead of sending the complete information to the LUS.

CMU-relevant identification data are also provided by the CD, and thus a respective parser request is made. Finally, the port of the HTTP server is requested and encapsulated in a suitable attribute.

Registration

Now, the plug-and-participate module has all the information necessary to join the community. It prepares the attributes by encapsulating the information into special Jini attributes - so-called `XMLEntry` objects that simply consist of the above XML-formatted data.

As a design decision, all functions are registered as single services in order to gain an abstract function-oriented view. This ensures completely independent modules, and thus it forms the second fundament of the horizontal integration apart from the abstract FCM concept.

The join process is rather simple comprising the preparation of `ServiceItem` objects (one per function) as already shown in section 2.3.2.2 (especially code example 3) and finally the invocation of the `register(..)` method of the LUS proxy reference (see therefore also section 2.3.2.2, code example 4). Such a registration, however, must consider the constraints given by the RA regarding the registration with a dedicated LUS only, or additionally a registration with further LUSs. In the first case, the Jini unicast discovery protocol is used to retrieve the LUS reference of the specific LUS, while in the second case, the references are retrieved from the discovery component of the plug-and-participate module.

Evaluation of the Registration Acknowledgement

After the registration process, the returned `ServiceRegistration` object is evaluated and the contained lease is extracted, which is passed to the lease handling component. `ServiceRegistration` objects are furthermore stored in a list in order to allow appropriate modifications of registrations. Moreover, the entire data prepared for the registration are also stored so that the functions can be registered with LUSs appearing during runtime of the CMU (if allowed).

As an acknowledgement of the registration process, the plug-and-participate module finally returns the extracted function-related information to the RA. In this way, the RA learns about the functions it represents in the PABADIS community.

5.3.3.3.2 Robustness Facilities

The self-maintenance procedures are performed implicitly in the plug-and-participate module. In this reference implementation, this mainly involves the lease renewal processes realized by specific renewal manager instances. Essential features of such renewal managers are facilities to store the leases and to determine the shortest lease duration of these stored leases. The lease renewal manager instance basically waits until a lease renewal process is due (defined by the shortest lease duration), and subsequently the renewal process is launched.

Storing leases simply entails holding them in a list. Based upon this list the renewal manager periodically determines the shortest lease duration, which indicates the time to wait until the leases must be renewed. After the renewal process, a new shortest lease must be determined. The entire procedure

runs in a simple loop in a `Thread`, and is only interrupted if a new lease is added to the list. In this case, a “new” shortest duration must be determined.

The renewal process itself is rather simple - it refers to the invocation of the `renew(..)` method of each lease object stored in the list. It expects the duration that is requested for the new lease period, which is in PABADIS given by a standard lease duration. This duration can be adapted by the RA, if it invokes the `adaptLeaseDuration(..)` method given in the interface shown in figure 15, line 2. Invoking this method causes the lease renewal manager instance to immediately apply this new duration to all the stored leases, and thus to start the renewal process with an adapted lease duration - the new lease duration of this CMU is immediately published to the community.

5.3.3.3.3 Attribute Handling

Attributes in PABADIS are the main facility to describe functions in the community. Such attributes might change over time. Also, in automation, the provided functionalities are usually described by specific attributes, and thus the abstract interface offers facilities to modify the functionality descriptions (see lines (3) - (5) in figure 15). PABADIS’s Jini-based reference implementation forwards the requests via the abstract interface methods to the Jini counterparts, which are defined in the responses of LUSs on service registrations - the particular `ServiceRegistration` object is determined, the respective method is invoked and the modified attributes are forwarded to the LUS.

PABADIS function attributes are defined in `XMLEntry` objects, which in turn carry `XMLObject` instances finally storing the data. This allows a generic and efficient description of functions, which is rather state-of-the-art in automation, but not in plug-and-participate.

Remark: For example, Jini allows arbitrary objects for attributes. The objects must have the supertype `Entry`. Jini’s matching facility, however, is based on the comparison of bytes of these objects - attributes are not restored in order to conserve resources.

The attribute format is furthermore the essential basis for an efficient matching algorithm, which itself is the fundament of the function search facility described in the following section. The matching is given in section 5.3.3.3.7, where also detailed information regarding the `XMLObject` infrastructure can be found.

5.3.3.3.4 Function Search

Automation systems require a facility to determine available functions in the community. This facility is defined by the `lookup(..)` method of the interface. In the PABADIS reference implementation, a request is directly mapped to Jini’s `lookup(ServiceTemplate templ, int count)`method, which provides at most `count` service references matching the information specified in `templ`. The `SearchPattern` object, which is passed to the `lookup(..)` method of the automation plug-and-participate interface, is converted to a Jini-compliant `ServiceTemplate` object and finally forwarded to the LUS. If the LUS finds matching services, it responds with a `ServiceMatches` object containing an array of `ServiceItems`. The plug-and-participate module subsequently evaluates the responses and creates suitable PABADIS-compliant `Matches` objects, depending on the service type. In particular, PABADIS defines/allows four service types corresponding to the proxy interfaces: `FunctionProxy`, `AgencyProxy`, `TraceProxy` and finally the LUSs. With respect to these types, the returned services are encapsulated into `FunctionMatches`, `LUSMatches`, `AgencyMatches` or `TraceMatches` and finally stored in an array. If the number of found services is less than `count`, the plug-

and-participate module requests further LUSs (if existent) for the demanded functions until either `count` is exceed or no more LUSs are known to the discovery component. Moreover, if functions are registered with more than one LUS, and thus found more than once, such copies are eliminated from the result set. The collected `Matches` objects are returned to the requesting client. This abstract description of a function search requests especially demonstrates the decoupling of PABADIS and any underlying plug-and-participate technology - the implementation converts PABADIS related data structures to such of the used plug-and-participate technology, and maps the request to suitable methods.

An important fundament to this function search facility is the comparison of requests with stored service registrations. In PABADIS, the entities are usually registered with LUSs by specifying XML-formatted attribute sets - the information is obtained from the CD of the CMUs. The counterparts, namely the search patterns of requests, are basically provided by PAs that in turn extract the patterns from their Work Orders - the requests are therefore also XML-formatted. If these requirements and input data are compared with the attribute matching facilities of Jini or other plug-and-participate technologies, this shows a gap between the requirements and the existing algorithms. Plug-and-participate technologies often do not provide XML-based comparison facilities. This problem led to a more suitable matching algorithm, which is shown in section 5.3.3.3.7.

The function search concept described in this section shows the active method for obtaining an overview of the function community. Planning systems, however, require an always up-to-date view of the infrastructure, and thus need to be informed immediately if any change occurs - a passive information system is required. Such a mechanism is defined in the abstract interface, and its Jini-based reference implementation is described in the follwing section.

5.3.3.3.5 Participation in the Passive Information System

The passive information system allows entities to be informed if changes in the infrastructure occur. It defines two fundamental parts: The active part tracking the community and dispatching the events, if any occur, and passive entities that wish to be informed. The latter must register with the active component in order to indicate their interest and to specify the events of interest. Such a registration requires a `ServiceTemplate` object defining the functions of interest, the event type specifying whether a function of interest joined or left the community or simply has changed its attributes, and a reference to an appropriate event listener. The information is passed to the plug-and-participate module via the `remoteEventNotifier(..)` method of the abstract interface - in a technology-independent fashion as a `SearchPattern` object and an `EventNotifier` instance. The registration itself is simply the invocation of the `notify(..)` method of LUS proxies retrieved from the discovery component. The received `EventRegistration` objects are then evaluated, and the extracted leases are passed to the lease renewal manager instance.

The second aspect apart from the registration is the delivery of the events: Plug-and-participate systems are generally designed for non-mobile environments, while in PABADIS usually mobile PAs (as entities of a distributed planning system) register for remote events. This impacts the delivery process, since events received at a local CMU must be delivered to a PA even if it is migrated to another CMU. In particular, each registration therefore is annotated with an identifier pointing towards the component that made the registration - if a PA is affected, it is checked whether the PA is still at the CMU. In this case, the event is delivered; otherwise, the event delivery is postponed. If a PA wants to move to another host, it has to set a certain flag, which indicates that the PA cannot be contacted locally. Incoming events for this agent therefore must be queued. If the PA arrives at its new host, it re-regis-

ters for remote events locally at the new host, cancels the registration at the old host and requests any postponed events.

This concept is further described in [74] and [76] as an outcome of the editor's work in this particular field. Such an extension of the remote event facility for mobile entities has been developed during the author's investigations for the PABADIS plug-and-participate facility in order to support PAs in their planning mechanisms, which requires an always up-to-date view. Unfortunately, this concept be cannot directly manifested in an abstract automation-related concept, because it depends on the particular automation solution and on the features of the plug-and-participate technology used.

5.3.3.3.6 Discovery

Although discovery is not defined in the abstract automation-related plug-and-participate interface, it is an essential feature of the Jini-based reference implementation for PABADIS - almost all facilities described rely on the knowledge about LUSs and their LUS proxies.

The discovery process is completely transparent for users, because it is implicitly started with the initialization of the plug-and-participate module: The Jini `LookupDiscovery` class is instantiated, starting the Jini multicast discovery protocol and finally switching to the announcement mode. A `DiscoveryListener` instance is installed, which receives the information, when a LUS is discovered. Each operation that needs access to LUS proxies retrieves them by invoking the `getRegistrars()` method of the `LookupDiscovery` instance.

5.3.3.3.7 The Matching Algorithm

In addition to the directly plug-and-participate-related facilities the framework requires efficient matching mechanisms. They must be compliant with up-to-date description technologies in the automation field, which are mainly based on XML. For example, in the PABADIS concept, the function search is strongly based upon XML-formatted attributes. The function search therefore requires a specific matching algorithm that is suitable for XML comparison. Moreover, function search basically must provide several features like exact, partial and range matching based on XML-formatted attributes.

Remark: Range matching determines whether a certain value is within a given interval. Partial matching, in contrast, defines a facility that allows for comparing only parts of an attribute. In particular, an entire template is matched to an attribute, while the attribute can consist of more information than the template.

Except for the exact matching feature, common plug-and-participate technologies cannot satisfy these requirements, and thus their matching facilities must be extended by these additional features. This section is divided into two special parts: First, the prerequisites in terms of necessary data structures will be sketched, and, based upon these structures, the algorithm will then be explained.

Both aspects are an important contribution in the field of automation and plug-and-participate, because they close the gap between the referred fields: While the field of automation more and more relies on XML-formatted descriptions, plug-and-participate does not necessarily cope with such descriptions. Further information on the basic facilities such as data structures used and the matching algorithm itself are published in [24] and [27].

Data Structures

In the PABADIS reference implemention, an algorithm has been developed and implemented that is based upon the comparison of so-called `XMLObject` instances. Such objects reflect the XML structure of function descriptions as well as orders: Leaves of an XML tree are encapsulated in `XMLObjectLeaf` objects having the following structure:

```
public class XMLObjectLeaf extends XMLObject implements
                                              java.io.Serializable{
   //the value of that leaf
   public String value = "";
   . . .
}
```

CODE EXAMPLE 14. The class `XMLObjectLeaf` representing leaves of XML structures

Intermediates node are implemented by `XMLObjectNode` objects that carry an array of `XMLObject` instances representing the children of the particular node. These child nodes stored in the array are in lexicographical order with regard to their tags. The `XMLObjectNode` class has the following internal structure:

```
public class XMLObjectNode extends XMLObject implements
                                              java.io.Serializable{
   //the child elements
   public XMLObject[] elements=new XMLObject[0];
   . . .
}
```

CODE EXAMPLE 15. The intermediate nodes of an XML structure represented by the `XMLObjectNode` class

Finally, the supertype of all is the class `XMLObject`:

```
public class XMLObject implements java.io.Serializable{

   public String tag = ""; //store the tag of XML object

   public String[] attributeNames; //store the names of the attributes

   //store the values of the attributes in a String array
   public String[] attributeValues;
   . . .
}
```

CODE EXAMPLE 16. The (super) class `XMLObject`

Such objects are easily serializable as well as de-serializable, because of their simple structure. It especially ensures a compact data representation that saves memory and allows efficient matching algorithms.

The Matching Algorithm

The referred data structures and resulting objects are the input for the algorithm described in [27]. This algorithm relies on templates and attribute sets - templates refer to the request patterns, while an attribute set refers to the stored functions.

As the first step, the algorithm checks whether the tag of the template's root node is contained in the list of attributes - this search process is based on a binary search, which is possible due to the lexicographical order of the attributes regarding their tags. If the tag is found, and thus a possibly qualified entry, the next level of both, the template and the attribute, is checked. This level-wise test is done recursively: Each tag of child nodes of the current template node is tested, whether it can be found in the corresponding child node array of the qualified attribute node - this search process is implemented recursively for each level until either the structures do not match or a leaf of the tree structure is reached or wildcards are found. The algorithm is illustrated again in the following example:

Template	**Attribute Set**

`<function>`
`  <name/> drilling`
`</function>`
	`<function>`
	`  <name/> drilling`
	`  <parameterSet>`
	`  <parameterSet.PABADISname/> noname`
	`  </parameterSet>`
	`</function>`
	`<HTTP/> 8080`
	`<SCADA/> true`

TABLE 6. Example for matching: here, partial matching.

The first step in the algorithm checks via binary search, whether the tag `function` of the template can be found in the attribute set comprising the tags `function`, `HTTP` and `SCADA` - `function` is found as the first entry in the list of attributes. Now, all child nodes of the template must be tested against the attribute having the tag `function`. In this case only the tag `name` exists, which is searched in the list of tags of the attribute node. This level stores the tags `name` and `parameterSet`. Of course, `name` is found via binary search, and simultaneously the recursion stops, because a leaf is found - both leaves contain the same value and thus the matching returns `true` - the partial match feature found a matching attribute.

The last aspect to be tackled regarding the matching algorithm is its complexity. The starting point of the evaluation is the number of templates, *k* - for each template, the entire algorithm must be performed. As the first step in the algorithm, the tags of the templates must be found in the list of attribute tags - a binary search is applied that can be estimated by a complexity of *log(l)*, where *l* is the number of subtrees/child nodes at each level. Finally, the algorithm is recursively called *h(tree)* times, where *h(tree)* describes the depth of the tree. This results in the following estimation of the complexity:

$k * \log(l) * h(tree)$

In the worst case scenario, *h(tree)* converges to *n* with *n* = number of elements in the tree - this occurs if each node has only one child node, and thus the XML structure degenerates to an arbitrary list. In this case, however, the effort for the binary search is constant, because only one comparison step has to be made. If nodes have several child nodes, the depth of the tree decreases and finally converges to 1 if all nodes are at the first level - in this case the complexity of the binary search is *log(n)*.

Both described scenarios simplify the aforementioned formula:

$O(k*n)$ in the worst case and $O(k*\log(n))$ in the average case, with k = number of templates and n = number of nodes in the attribute trees.

Comparing these complexities with Jini's counterpart, this algorithm relying on XML tag-based trees is more efficient and provides more features. Jini's matching complexity can be estimated as $O(k*n*m)$, where k = number of templates, n = number of attributes and finally m = number of template bytes. Jini's matching algorithm can be abstracted to the following code fragment:

```
for each template [i]
  for each attribute [j]
    for each template[i].byte[k]
      result &= template[i].byte[k] == attribute[j].byte[k]
```

CODE EXAMPLE 17. The principle of the Jini matching algorithm

The matching algorithm described in this section, of course, is neither bound to PABADIS nor to Jini or any other technology - it is suitable for incorporation into almost all technologies and furthermore suitable for automation in general. For example, description technologies in automation are more and more XML-based, and thus appropriate matching facilities are required for assigning tasks of orders to suitable device functions. This XML-based matching algorithm especially results from this necessity. It furthermore overcomes the shortcomings of current plug-and-participate technologies, which is a further result of the author's investigations in the plug-and-participate field for automation.

5.3.3.4 A UPnP-Based Implementation of the Automation-Related Plug-and-Participate Interface

While the previous section described a Jini-based reference implementation for the automation-related plug-and-participate interface as used in PABADIS, this section provides a hypothetical reference implementation based upon UPnP. This means, the required facilities defined in the interfaces are investigated under consideration of corresponding UPnP mechanisms. On the one hand, this shows the independency of the interface from any specific technology. The resulting reference implementation simply rebuilds the PABADIS plug-and-participate feature using UPnP. On the other hand, this UPnP-based reference implementation likewise proves the independency of PABADIS from any specific plug-and-participate technology. This mutual independency can furthermore be explained if the specific facilities of the two technologies are kept in mind - UPnP and Jini define the extremes in plug-and-participate.

A primary goal of the automation-related plug-and-participate interface and PABADIS is their opennes to any (compliant) technology. A secondary goal is defined as the broad investigation in this field instead of concentrating on only one technology. In this way, a certain spectrum of technologies and

solutions is covered, which evaluates the possible fields of application, and most importantly evaluates a possible acceptance of PABADIS. However, especially from the target area of PABADIS - legacy automation devices - results the necessity of a plug-and-participate technology suitable for limited device. Although UPnP and Jini can be used to implement the automation-related plug-and-participate interface, they do not meet the limited device requirements. This is due to their inherent drawbacks, and a more suitable solution for limited devices in the automation field is required. Such a technology should consider these drawbacks as well as the advantages of both solutions.

5.3.3.4.1 Community Join

In the same way as described for the Jini-based reference implementation, the entities must join the community in order to make their facilities available to other entities. With respect to PABADIS, the CMUs must offer their functions to the network - in contrast to the Jini-based reference implementation, a CMU joins the network as a whole, offering several services. This results from UPnP's device orientation, and thus treating a CMU as a root device is reasonable. It consists of services representing the functions. The community join, once triggered by invoking the `joinCommunity(..)` method, comprises mainly the same operations as the Jini counterpart, namely the extraction of data from the CD and its preparation. Subsequently, the data must be advertised to the community. The data preparation in general does not differ from the Jini-based reference implementation, except that the data are simply provided to the community via UPnP's description facility, so that they can be requested from client entities via HTTP. This strongly differs from the Jini counterpart, where the description information is sent to the LUSs during the join process.

The advertisement process finally means the transmission of messages for the CMU - 3 messages for the root device (the CMU) are sent, and one for each offered service. Such messages contain the required information about the existence of the device and likewise its services, and they furthermore carry URLs where description data can be retrieved.

5.3.3.4.2 Robustness Facility

A UPnP-based reference implementation also has to cope with robustness and reliability issues in the same way as described for the Jini-based implementation: UPnP advertisements are annotated with a dedicated validity duration indicating when an advertisement becomes invalid. If the device or service, however, wants to remain in the community, it has to re-advertise its availability and a new validity period - the device or service periodically sends the same advertisement messages as used for the initial advertisement to the network. In this way, each control point can maintain its own view of the community. This validity duration, of course, can be also adapted through the `adaptLeaseDuration(..)` method. A new duration is then immediately applied to the system by triggering the advertisement process.

5.3.3.4.3. Attribute Handling

Attributes describe the facilities of PABADIS CMUs and their functions. In Jini, they are published to the LUS, while in UPnP, they are provided via the description mechanism: A CMU offers its attributes via HTTP to the community. Interested parties can download it from the URL given in the advertisement messages. Each description is related to a certain device/service, and thus a direct relation to the validity duration of advertisements can be observed - descriptions have the same validity as their advertisements. If an attribute must be changed this automatically changes the description - therefore,

the advertisement must be revoked and immediately re-advertised in order to announce the changed attribute. These attributes can then obtained from the URLs given in the new advertisements.

Taking this into consideration and keeping the methods given in the abstraction interface in mind, a direct correspondance of the described mechanisms to them is obvious - if a new attribute is added - by invoking the `addAttribute(..)` method - the advertisement is revoked and re-advertised; if an attribute changes, the same procedure is performed as well as if an attribute is deleted - triggered by the invocation of the `changeAttribute(..)` or the `removeAttribute(..)` method, respectively.

5.3.3.4.4. Function Search

In the same way as in the Jini reference implementation, invoking the `lookup(..)` method of the interface returns information about suitable functions available in the community. The UPnP-based implementation uses the UPnP advertisement mechanism to request the infrastructure for demanded functions. A second alternative can be provided, if the plug-and-participate module tracks the infrastructure and stores all CMUs and their functions in a list that is maintained by the periodic advertisements of the devices and services. If a request is received, it can be handled locally based on the given data.

Regardless of the alternative used, the `lookup(..)` method finally returns the available functions to the requesting client. This, however, requires that the URLs retrieved via the advertisements are used to download the descriptions of the CMUs and their functions in order to provide these descriptions appropriately to the requesting clients.

5.3.3.4.5 Participation in the Passive Information System

The UPnP infrastructure implicitly defines a passive information system indicating when events occur, but it requires suitable management facilities: Because of the periodically sent multicast advertisement messages, each control point is always informed about available functions and CMUs in the community. Each control point must therefore maintain a list containing this information; otherwise it does not become aware of leaving CMUs and functions, if they do not revoke their advertisements. In contrast, newly arriving functions and CMUs are detected only, if the list is checked for a received advertisement. Changed attributes can be detected if CMUs or functions revoke their existence and subsequently re-advertise their availability and announcing a new description. Such a behavior recommends the second alternative for function search sketched in the previous section. However, this results in a high degree of redundancy in the system - and a waste of memory. Specific management mechanisms are required, such as a list of existing CMUs and functions, and a mechanism that checks the list if advertisements arrive. Furthermore, this mechanism must check, if the validity of advertisements is exceeded. Each CMU, therefore, has to track the community and to use the periodic advertisements for maintenance of the function and CMU list. Such management facilities, on the one hand, increase the robustness of the entire PABADIS system, because no single point-of-failure exists. On the other hand, this robustness only results from a high degree of redundancy that, of course, wastes memory, which is usually not available at limited devices of the automation field.

5.3.4 Benefits of Plug-and-Participate in Plant Automation

Section 5.3.3 provided a detailed evaluation of plug-and-participate in plant automation with respect to the MES level. Results are a dedicated automation-related plug-and-participate interface shown in figure 15, and two basic reference implementations - one based on Jini, and the other based on UPnP.

These features, in turn, must be investigated whether they ensure the expected benefits for the automation world. This section therefore provides examples for using the plug-and-participate concept in PABADIS, especially dedicated CMU facilities are described. The major focus in this respect is on the plug-and-participate impact on planning system facilities - mainly the behavior of PAs in a PABADIS system is of interest. The required plug-and-participate features for agents were provided by the author of this thesis, while the planning facilities result from intensive cooperation within the PABADIS team.

Aside from the particular PA behavior, this section furthermore demonstrates specific functionalities provided by dedicated CMUs. These CMUs and specific components likewise mainly result from the author's work during the PABADIS development and the integration phase, and therefore they essentially follow requirements of PABADIS systems. The results are not bound to PABADIS only, but define more general approaches: They can be abstracted to general solutions in the automation field such as an alleviated SCADA architecture set up or robust trace facilities. Such examples could result in respective innovations in the automation field which are finally contributed by this thesis.

Plug-and-Participate Impact on PABADIS Planning Facilities

PABADIS provides a highly distributed planning mechanism - mobile Product Agents are the main entities of such a distributed planning system, which are responsible for controlling the product manufacturing. Along with the planning algorithms themselves, a PA relies on two basic prerequisites: an order describing the tasks to be done and an overview of the community, and thus information about available functions. While an order is provided by the PABADIS-Agency, the infrastructure overview is provided by the underlying plug-and-participate layer. The PA's planning module "simply" assigns processes specified in the tasks of an order to available functions, which neither shows any innovation nor particular benefit - such behavior is the state of the art. PABADIS, however, revolutionized this process by introducing flexibility and adaptability into the algorithms. PAs do not rely on static plans; they can adapt their plans according to changes in the community. This capability is based on a dedicated scheduling algorithm that fundamentally relies on plug-and-participate mechanisms. In particular, the algorithm assigns processes to functions. It therefore requests the plug-and-participate system for suitable functions using the `lookup(..)` method of the abstract automation interface. The schedules resulting from this function assignment procedure can be assumed as up to date, at least for a particular moment - if changes occur, the plan must be adapted. A PA becomes aware of changes via the plug-and-participate-based passive information system, which is the basis for an alleviated plan adaptation. Such an adaptation mechanism is furthermore based upon the distribution of the planning tasks, namely each PA only considers its particular order and its created plan. Changes in plans and schedules therefore only affect a single PA, and thus remain local instead of diffusing through the system. In addition to efficiently provided information about changes, also the efficient application of these changes to the plans is ensured due to the decreased complexity of plans - a PA only needs to schedule its own order without having to consider other orders. The efficiency and flexibility gained describe the first innovation in this area, while the second innovation is defined by a flexible infrastructure provided by dedicated plug-and-participate technologies. Both innovations together result in a revolution-

ary planning facility, which is already tested on a first testbed called the Fischertechnik demonstrator. It is a toy-like plant infrastructure as shown in the following figure:

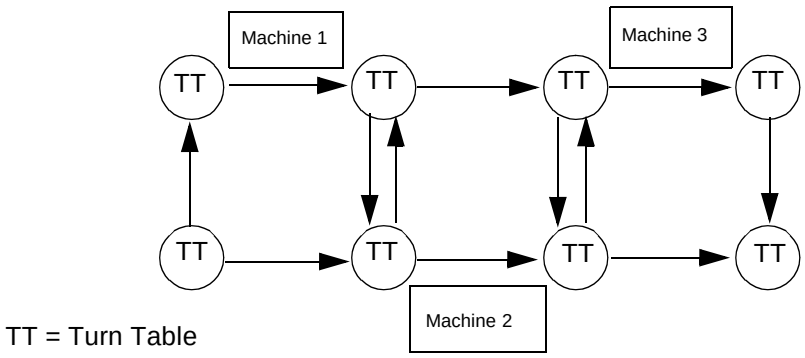


FIGURE 17. The structure of the Fischertechnik Demonstrator

Machines are represented by three CMUs that always offer three identical functions. A Routing CMU and a Transport CMU complete the plant infrastructure, and a PABADIS-Agency provides the connection to an ERP system. If the PABADIS-Agency releases PAs, they immediately start their order evaluation - they obtain an overview of the plant infrastructure including the dedicated transport topology. Subsequently, PAs concurrently perform scheduling and resource allocation for given CMUs, and finally navigate through the system. Assuming a stable system, and evaluating the resulting schedules of CMUs and PAs, a rather optimal throughput of the demonstrator infrastructure results compared with schedules calculated by conventional centralized planning systems. However, if the demonstrator system gets rather instable due to failing CMUs or transport functions, the innovative PABADIS planning facilities allow a robust and nearly optimal behavior, while central planning systems cannot adapt to these events efficiently. Strictly speaking, Product Agents receive such events via the passive information system and can immediately react to the new situation. In such potentially instable system, the planning mechanisms of PABADIS show essential advantages compared with central planning systems. The same observation results if the order of a PA is changed: In this case, a PA can immediately incorporate a new or changed order into its planning processes.

Concluding such a fundamental and essential observation shows that turbulent environments do not cause replanning of entire plans as common in centralized systems, but stay local only at the concerned PAs and their plans. These specific plug-and-participate-based planning facilities especially result from strong cooperations with involved PABADIS partners and the thesis' author - the evaluation of plug-and-participate capabilities resulting in flexible machine networks and the distribution of MES facilities culminate in the presented mechanisms. Such innovations in planning and MES in general are not the only facets, plug-and-participate also allows several useful and innovative features in plants in a more efficient, but also standardized way. With respect to PABADIS and its plug-and-participate facility, different mechanisms, concepts and CMUs have been developed that strongly rely on plug-and-participate. These CMUs/components are briefly sketched in the following sections, which documents further results of the author's work in PABADIS.

PABADIS Transport CMU

PABADIS models the transport capabilities as arbitrary CMUs providing dedicated transport functions. As in general plant automation concepts, also in PABADIS transport is a crucial point, because it defines the underlying topology, which connects the machines. Such topologies must also be flexible and adaptable, and furthermore, they must allow a certain flexibility of the system. For example, if a transport system can easily be extended without any essential re-configuration, this will ensure a high degree of flexibility, and thus will result in appropriate benefits for the plant.

PABADIS Transport CMUs therefore can easily be integrated into a plant topology. This usually means changes in the infrastructure, which causes PAs to adapt their paths through the plant according to the current topology as soon as they become aware of changes in the infrastructure. However, integrating transport nodes into an already existing topology or even the setup of a new topology requires special attention due to the specific architecture of transport functions: They internally know their provided transport routes, e.g., transport from point A to point B, where A and B are local points in the CMU. PAs, in contrast, only know about CMUs and their functions as nodes in the topology. Dealing with this problem results in a dedicated mapping of local Transport CMU points to global CMUs. The mapping information, in turn, is published by the plug-and-participate system so that each entity can learn about the plant topology. In PABADIS, this mapping is solved by a suitable GUI application from which CMUs can be assigned to the local points manually. Changing routes in the topology, because of either additional or reduced capacity, therefore simply requires a modification of the mapping information, which is then automatically published to the system as plug-and-participate attributes. If, for example, entities are registered for remote events pertaining to these attributes, they will immediately be informed, when changes in the topology occur. Such entities, which always need to know about the current plant topology, are PAs and the Routing CMU - while PAs need this information in order to check, whether their plans are still up to date, Routing CMUs rely on an always up-to-date view of the community in order to provide valid routes through the plant. This will be demonstrated in the following section.

PABADIS Routing CMU

The Routing CMU supports PAs in navigating through the plant. PAs usually know the starting point and the destination CMU of a transport task, but they do not know about the particular path through the system. For this reason, a PA requests a Routing CMU for a suitable route: A Routing CMU maintains a representation of the entire system topology as a graph and calculates routes based upon a modified Dijkstra algorithm. This graph is the main prerequisite for the algorithm and must be reasonably up to date - the plug-and-participate layer ensures the required up-to-dateness. For example, Routing CMUs query the plug-and-participate system for Transport CMUs and evaluate the provided paths and their connected processing CMUs. Based on this knowledge, the graph is created. Routing CMUs furthermore register for remote events regarding Transport CMUs, and thus ensure the required up-to-dateness.

Both CMU types together, the Routing CMU and the Transport CMU, provide a modular topology that is very flexible and efficiently adaptable to new requirements. The plug-and-participate system in this respect offers a standardized and homogeneous view of entities of the system. This fundamental basis of an innovative transport system allows a flexible and efficient setup of transport topologies as well as transparent modifications of the system such as increasing or decreasing transport capacities. Such a flexibility is reflected in the plug-and-participate system, and thus, interested entities are immediately

informed about topology changes. Based upon this information they can react appropriately, and thus ensure a robust, and mainly efficient system - PAs adapt their plans according to the changed topology, when they are informed via plug-and-participate facilities.

PABADIS-Agency - Plant Observer Facility

The support of PAs in their detailed planning of tasks and paths through a plant is the main objective of the PABADIS approach, which is essentially based on plug-and-participate patterns. Such patterns are the fundamental basis for an automatic adaptation of plans to the current plant topology - this is a major innovation in this field. A second aspect in this planning respect lies in supporting the Work Order extraction out of Manufacturing Orders, which is done at the PABADIS-Agency. The essential focus in this complex process is on the mapping of process descriptions given by the Manufacturing Order to functions provided by the community. These functions will finally be listed in the tasks contained in the created Work Orders.

For this reason, the PABADIS-Agency comprises a dedicated module, which maintains a list of all available functions and CMUs. It is called the Plant Structure Observer. This observer module retrieves its knowledge from the plug-and-participate system, where it is registered for remote events. If any event occurs, it is immediately informed, and the Work Order creation process can incorporate the modified plant structure into Work Orders. This way, it can be ensured that generated Work Orders can be successfully performed by the involved PAs - at least at the moment when the order is created.

This Plant Structure Observer module is furthermore used in a dedicated Observer CMU providing monitoring functionality for the community. Such an observer CMU alleviated the test phase of the PABADIS project, and finally improved its reliability and robustness. For example, during distributed tests, where several partners ran several CMUs at different locations, each partner launched an Observer CMU in order to monitor the system behavior. The CMU displays the infrastructure such as its registered CMUs and their functions, and even active PAs in the system can be monitored. In this way, the PABADIS system is extended by a useful monitoring tool for tests in order to detect inherent system problems and the misbehavior of agents, etc. The Observer CMU furthermore provides a reasonable testbed for the PABADIS Agency's plant structure observer module itself as well as a suitable demonstration platform of the system for interested parties.

In addition to these rather planning system related benefits, plug-and-participate systems also alleviate appropriate monitoring facilities and mechanisms for testbed observation. With the Observer CMU one specific representative has already been mentioned. Likewise an alleviated SCADA architecture setup and maintenance can be achieved by plug-and-participate-based concepts as well as robust tracer mechanisms. Both will be briefly sketched in the following sections:

PABADIS SCADA CMU

SCADA - Supervisory Control And Data Acquisition - reflects two directions: control of machines in terms of dedicated configurations and data acquisition for monitoring reasons. Both mechanisms require connections between the SCADA system and the machines to be either monitored or controlled. In legacy systems, such connections are usually manually established and the configuration for those relationships is also manually controlled. PABADIS incorporates a SCADA system, which is modeled as a dedicated CMU providing a SCADA function. This leads to two distinct parties to be clas-

sified: The SCADA system, on the one hand, and the CMUs which want to be supervised, on the other hand. Both must be connected efficiently. In PABADIS, this is realized by a special agent, a so-called SCADA PMA. It seamlessly and transparently establishes the required connections, which essentially rely on plug-and-participate features: The PMA requests the underlying system for SCADA CMUs in order to retrieve their particular configuration data. The second step is to request the system for CMUs, which want to be supervised by SCADA. Based on this knowledge, the PMA establishes the connection between the CMUs and the SCADA system, and the agent itself acts as a broker in this relationship as shown in the following figure:

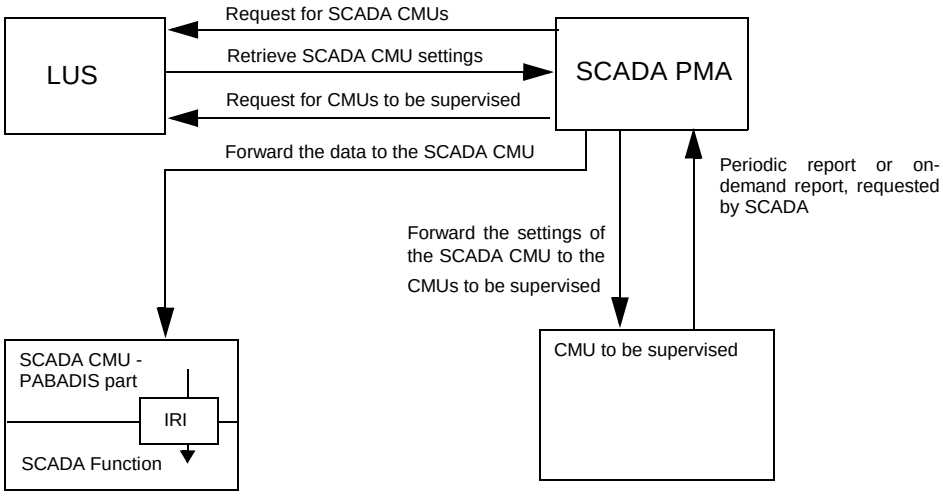


FIGURE 18. The PABADIS SCADA Architecture

This broker facility results from the transparency of the SCADA-related connections. If a SCADA CMU disappears, the processing CMUs must not be affected. They furthermore contact the SCADA PMA sending their periodic reports, while the PMA tries to connect another SCADA CMU in order to complete the architecture. The second aspect reflects the SCADA PMA - CMU connections. If a new CMU appears that wants to be supervised, the PMA is informed via the passive information system of the plug-and-participate layer. The PMA contacts the CMU and integrates it smoothly into the infrastructure - without any manual intervention. The same applies for CMUs leaving the system, since the PMA automatically removes them from the architecture.

Reflecting this briefly described PABADIS SCADA architecture and comparing this with legacy systems - this architecture marks a major improvement in efficiency, robustness and maintainability. The plug-and-participate mechanism is one fundamental aspect in this architecture.

PABADIS Tracer Module

Despite the rather plant- and production-related use cases, a further, more testing- and controlling-related concept will be briefly described, namely the tracer concept of the PABADIS system. It aims at tracking the system behavior in order to get an overview of actions that occur in the system. This covers actions in a local CMU scope such as initialization of modules, state transitions in the FCM, etc.

The tracer furthermore comprises a global tracing mechanism, which tracks global agent behavior, CMUs or general global entities. Such requirements result in two dedicated components: a local tracer module that stores local trace messages. This module furthermore forwards global trace messages to a global trace server - the second component in this architecture. While the local trace facility does not rely on any plug-and-participate feature, the connection to the global trace server is implemented via arbitrary plug-and-participate mechanisms. Such a global trace server is implemented as a plug-and-participate service, whose reference can be retrieved via a simple lookup request. Global trace messages, in turn, are forwarded via this reference to the server, transparently for the local module.

Such a tracing concept is reasonably stable: If a global trace server fails, the local module tries to find another server, which ensures a reliable architecture. Moreover, the use of plug-and-participate also alleviates the setup of the relationships without manual configuration. For the general concept of this tracer facility please refer [75] - the concept regarding the trace levels results from the cooperation of the team of the University of Marburg with involved partners from industry, while the approach to use plug-and-participate for global tracing results from the author's investigations in this field.

All concepts described in this section demonstrate particular use patterns of plug-and-participate technologies in MES-level tasks. They allow concepts with a high degree of flexibility and adaptability, as well as ensure the necessary robustness and transparency. Such gained benefits can be concluded to an increased efficiency of automation systems - the described concepts are suitable for today's turbulent markets because they are able to cope with quickly changing environments and demands. Plug-and-participate always played an important role and the concepts are mainly based upon this technology genre. It is usually the basic information provider for the algorithms and furthermore alleviates the setup of required system infrastructures.

However, the sketched concepts and mechanisms were basically related to planning and organization tasks, which can be classified as MES-related mechanisms. Integrated solutions furthermore require such facilities on each automation level, so at field level too. Control applications and control solutions need to be evaluated with respect to an increased flexibility and reliability provided by plug-and-participate technologies. This will be the topic of the following section, which simultaneously covers the second part of this chapter, namely plug-and-participate on the field level.

5.4 Plug-and-Participate on the Field Level

The major goal of plug-and-participate on the field level is rather similar to the MES-level counterpart: the provision of flexible and adaptable solutions. It will be achieved by redesigning automation applications that control appropriate field-level devices. This redesign and the possible introduction of plug-and-participate must consider general constraints of automation control applications such as realtime or deterministic behavior. The proposals and concepts given in this section therefore do not affect such critical parts of control applications, but instead aim at providing mechanisms for efficiently offering functionality, increase the added value and efficiency of machines, increase their functionality and provide facilities to alleviate the creation and setup of applications. Moreover, the proposals given are related to the entire field level, ranging from fieldbus systems via enhancements for distributed auto-

mation solutions like IDA and PROFInet to completely distributed automation applications offering fine-grained automation functions. All these applications show different aspects of plug-and-participate on the field level, and they are the basis for higher-level solutions like PABADIS and HMS. A second major aspect of these applications aside from their revolutionary design is their field of application: They are generally applied to automation devices directly, and thus must be suitable for limited devices, which results in a further motivation for limited device compliant plug-and-participate technologies.

5.4.1 Totally Distributed Automation

The basic objective of totally distributed automation is the enhancement of machine capabilities and their efficiency, and thus to increase the added value of these machines. Such enhancements also point towards increased flexibility and adaptability, especially with respect to provided applications, their creation and their architecture. For example, machines generally provide functionalities that are combined in control applications that finally control dedicated production steps. This results in a concentration of responsibility in applications for the processing of production steps - they are completely dependent from products and production steps. In other words, legacy automation applications are tailored to specific products and steps. Such depicted legacy systems furthermore require that the applications are pre-installed at the machines. Moreover, if a new product must be incorporated into the production, new applications need to be developed and installed - in advance and in addition to the construction plan of the product.

Automation applications in general rely on basic functions of machines that can be interpreted as function blocks such as those defined by the IEC 6-1499 standard. If these blocks are provided to an infrastructure allowing a free combination of them on demand will increase the flexibility and adaptability of machines as well as their efficiency, added value and functionality - a machine is no longer dependent from specific and pre-installed applications. Freely and on-demand combinable functions allow a high degree of flexibility, while statically defined legacy automation applications show massive deficites regarding flexibility, adaptability as well as a limited efficiency.

Both facets together - the provision of fine-grained functions and the possibility to freely combine them to control applications - define a new control application paradigm based on new control application generation methodologies. This new concept leads to two interlocked parts, namely the provision of functions in a flexible and efficient manner, and facilities and algorithms for their on-demand combination - the major focus in this section is on function provision. An example for this is given in [106]: In this thesis, written at the ETH Zurich/Winterthur (a partner of PABADIS), a robot has been developed and evaluated that provides fine-grained functions like movement operations, pick-and-place operations, etc. The authors of this thesis intended to provide a special PABADIS testbed, and thus the scope of this work essentially differs from that of this section, but it does give a reasonable example of the idea of this section.

Remark: The considered robot is a SIG PACK Delta Robot, SIG XR 33 Delta Robot based on a Siemens Cimetrix controller architecture running on a Dual-Pentium IPC with two OS: Win NT 4.0 and RTX. For more detailed information regarding the robot and the concept, please refer to [106] and [88].

While this robot is simply designed as a PABADIS CMU offering dedicated functions based on Jini, the concept here aims at providing a more abstract concept offering fine-grained functions via plug-and-

participate technologies to a community. Within this community, the functions will simply be handled regarding their provided functionality rather than as dedicated functions of particular devices. Moreover, the concept is neither bound to robots only, nor to Jini, nor to any particular plug-and-participate technology, but it covers the entire field of automation applications and devices under consideration of plug-and-participate features: The approach, especially the plug-and-participate related function provision, comprises several topics and steps. In particular, a vendor has to identify particular fine-grained functions of its machine. Based upon this information the functions have to be described in a standardized way considering specific semantic patterns and rules. This leads to the definition of an ontology specifying description patterns as well as the semantic of provided functions. These functions must be provided to the infrastructure - plug-and-participate facilities in conjunction with appropriate semantic approaches efficiently offer the functions so that upper layer parts can access these functions regarding their provided features. The resulting infrastructure comprises these abstracted functionalities in a device-independent way and is therefore the basis of a system finally defining a revolutionary methodology for application design and creation. One specific methodology in this respect is classified by the statement “The order is the application” - it is described in a research project proposal submitted to the EC. This approach is called CODING - the specific contribution of the thesis’ author to this concept is the idea of the fine-grained function infrastructure based on plug-and-participate technologies and function abstraction as the basis for an innovative control application creation methodology. Further concepts, that potentially benefit from such infrastructures of fine-grained functions are IDA and PROFInet. This enhancement will be described in the following section.

5.4.2 Plug-and-Participate Enhancements for IDA and PROFInet

The basic concepts of IDA and PROFInet have already been given in sections 3.2.2 and 3.2.3. Both approaches are basically related to the field level rather than to higher levels, and define methodologies and architectures for control application creation. Keeping these concepts in mind, both approaches combine sets of functionalities to applications having specific architectures. This relies on different views of those function sets: IDA generally designs applications with respect to the required functionality regardless of specific devices, while PROFInet designs applications for specific machines. These different views basically impact the application structure. Abstract IDA Resources represent the functions and are finally mapped to dedicated IDA Blocks. In contrast, PROFInet basically reflects the machine structure within applications and maps appropriate object representations of machine facilities to dedicated objects of the application level.

Two basic parts can be identified from this brief survey of the technologies in this thesis, namely the structure of the abstract applications and the mapping of physical functions to their object representation. This latter mentioned mapping facility can be further identified as the crucial point in the solutions. In both architectures it is done manually, either via catalogue information statically defining the mappings like

```
<mapping from = "Basic IDA Block" to = "IDA Resource"/>
```

PROFInet, in contrast, connects the components using special engineering tools. This also comprises the download of the required configuration information via a mouse click.

According to the objectives of IDA and PROFInet - alleviation of the creation of automation objects based on existing function modules - a reasonable enhancement is obvious: If automation modules are provided by plug-and-participate facilities, as described in the previous section, and if the mapping

is realized via dedicated, plug-and-participate-based function search processes, both application architectures will gain enormous advantages with respect to saved resources during the engineering processes. Likewise this proposed concept increases the robustness, reliability and transparency of automation applications. Moreover, a more strict separation between physical devices and the application logic can be achieved, especially for IDA. This is furthermore in correspondance to the concept of totally distributed automation. For example, the plug-and-participate-based infrastructure provision requires semantic for such mapping: It covers the main prerequisite for the infrastructure provision of IDA and PROFInet applications, which alleviates the mapping of required resources to provided functions. In particular, plug-and-participate solutions such as the proposal for IDA and PROFInet as well as that sketched for the CODING concept are basic innovations in this field - they will allow for more efficient automation concepts, which usually increase the benefits for users and likewise for the vendors. However, such mapping always requires semantic for alleviated and automated infrastructure setup, which has not been tackled yet - it is topic of the following section, which describes an approach for fieldbus-based system infrastructures created by specific plug-and-participate facilities utilizing ontology concepts.

5.4.3 The Semantic Fieldbus Approach

Plug-and-participate concepts aim at providing service - service user relationships in a transparent fashion. This mainly refers to the function search processes. Clients, in general, search for dedicated services providing a specific functionality. If these rather pragmatic search and infrastructure setup processes are abstracted, an essential semantic-based concept results. The assignment of services to service users relies on the semantics specified in both parts, in the services as well as in the clients. In legacy applications, this semantics is generally given by specifying the required service/function during the development - in “next generation” applications, this might not be the case if the requirements result during the application runtime, and thus they cannot be incorporated in advance. This leads to the necessity of a formal description of the special semantics of applications in a suitable way - for example in an ontology.

Remark: The term ontology originally comes from philosophy, where it means a "theory about the nature of existence, of what types of things exist; ontology as a discipline studies such theories" (see [6], paragraph 2). In the context of computer science, ontologies can be defined in the following way: "An ontology is a document or file that formally defines the relations among terms." (see also [6]) - they provide a well-defined terminology throughout the information in a certain system (see [80]).

The concept of ontologies can be used for dynamic assignment of service - service user relationships, if requirements arise during runtime of systems. This especially alleviates the setup of infrastructures in general, and moreover provides a high degree of flexibility and adaptability in spontaneous networks. For example, if a relationship is broken, the affected participants can chose another suitable partner under consideration of defined semantic constraints.

The same arguments apply to fieldbus systems in conjunction with plug-and-participate facilities: Although fieldbuses are rather static networks, they can benefit from plug-and-participate mechanisms in several ways. Setup procedures can be tremendously alleviated, and fieldbus networks could be more flexible than they are today, if the relationships between fieldbus nodes are established via plug-and-participate concepts.

Remark: Fieldbuses originally come from process engineering in order to reduce the cabling effort at field level. Now, fieldbuses are an established standard in industrial automation (see also section 3.2.2 and 3.2.3, specifically the motivational paragraphs). Further information regarding fieldbus systems are in [86].

Today's fieldbus systems are basically configured manually in complex configurations procedures. For example, the nodes are identified and connected via complex tools, and related linkage information is configured manually. This can be alleviated by plug-and-participate; for example, sensors and actuators can be connected automatically, if both fulfill specific attributes. Such attributes, in turn, have to follow a certain semantics. A possible approach for establishing dynamic connections among entities can be relatively simple, namely the provision of a description by a sensor depicting the data it measures. This description is finally sent to the fieldbus via broadcast. Actuators, in turn, can receive and evaluate this information, and if their defined search patterns match the description, they can "connect" with the sensor. This leads to self-configured fieldbus infrastructures established in a spontaneous fashion. Moreover, the decision as to whether a relationship is established depends on the content of the description and on the features provided. This rather UPnP like concept, however, relies on an innovative facility - it needs formal description patterns and semantics. Such a requirement essentially fits to the objectives of the ontology concept, which is finally the recommended facility for that purpose. This results in a system behavior pattern driven by the semantic of the nodes; a relationship is established if and only if the prerequisites and requirements of the participants match.

A structure like this shows essential parallels to the semantic web approach (defined in [6]), where mechanisms are described, which alleviate the decision whether a certain WWW page is of interest to an agent's owner. Applying this concept to fieldbuses defines sensors as counterparts for the web pages, while actuators are comparable with the agents. These fundamental similarities between both approaches justify the name "semantic fieldbus approach" for the concept. In particular, the specific semantics of this approach is manifested in dedicated ontologies. They define the patterns for setting up relationships among entities - they offer their capabilities in an appropriate way using the information given in the ontologies. Client entities therefore can easily decide whether a capability matches their dedicated patterns describing the requirements of the entities - even if the functionality/demand appears during runtime.

Although this semantic fieldbus approach is defined for fieldbus systems, its general principles can be easily adapted and applied to arbitrary plug-and-participate systems and plug-and-participate-based solutions: The mapping of requirements to prerequisites has been identified as the crucial point in such systems - this mainly concerns the search process and matching procedures. A second aspect covers the description of requirements and prerequisites. Such descriptions are the basis for the matching as already shown in the algorithm given in section 5.3.3.3.7. For example, generic tag-based tree structures can be used as description facility for the semantic fieldbus approach, which allow very efficient and flexible matching algorithms (see also the `XMLObject` hierachy in section 5.3.3.3.7, and [24]). The general mechanisms of the concept are rather simple - entities announce their facilities using such tag-based tree structures; e.g., a sensor uses a dedicated structure for its data and simply starts to broadcast appropriately structured messages to the community. Actuators behave as clients and receive the issued data. They moreover use a pattern describing the structure of the required data in order to check whether the received message is of interest (please refer also to the matching algorithm given in section 5.3.3.3.7, because the same mechanisms are applied in principle). If the data match, a relationship is established in a spontaneous fashion. This means, the actuator evaluates the data, pro-

cesses its respective actions, and finally waits for the next message. This matching facility is with respect to the semantic of the patterns and messages - the search pattern of an actuator might match a message structure, although the tags do not match exactly; in such cases certain synonymous values, tags or expressions might match the request as defined in an ontology. An example of such an ontology can be found in [26], which moreover defines the structure of the messages as well as certain relationships. The definition of tags for that example application basically allows the alleviated setup of an infrastructure consisting of sensors and actuators, if sensors simply issue messages of the defined structure, and actuators match the messages against their patterns under consideration of the ontology semantics.

In addition to this rather innovative setup of fieldbus infrastructures (or more abstractly, service - service user relationships) based on dedicated sematic rules, the concept shows further innovative facets: If the messages and their structure are used as an addressing mechanism based upon a dedicated semantic specified in an ontology, a revolutionary mechanism can be observed. Legacy systems comprise a certain semantics - usually predefined - since sent messages always require an identifier of the target node. In contrast, the semantic fieldbus approach defines the message structure as the address: Interested (addressed) entities compare the structure with their patterns in order to check whether the message is of interest or not. This, of course, is innovative in the fieldbus area. The crucial points concern the definition of suitable ontologies and furthermore the application of the defined semantic in the matching procedures.

A further aspect that can be defined in such an ontology in addition to the syntax of the tags and possible synonyms, is the possible definition of a special semantic of the tags themselves. For example, the tags along the evaluation path from the root of a tag-based tree down to a value (stored in a leaf) can be interpreted as a procedure for processing of a specific value - a further innovation in this field.

If all these aspects are summarized, an innovative plug-and-participate as well as communication mechanism results, which is further described in [24]. This specific topic is presented there with respect to fieldbus device descriptions based on tag-based trees. The basis for these mechanisms is given in [26], namely the concept of the semantic fieldbus and an example ontology for a simple sensor system. The thesis' author and his colleagues evaluated this field in order to find possible solutions for semantic-based plug-and-participate mechanisms. Such mechanisms are especially motivated by fieldbus systems themselves and their necessity to alleviate the setup processes - the approach itself is more general and generic, and can be applied to the concepts described in previous sections. For example, the three main facets of the semantic fieldbus approach can be abstracted for general plug-and-participate solutions that basically rely on:

- the description of entities in a generic and well-structured format,
- the matching of requirements and prerequisites in order to establish relationships among entities and
- ontologies defining the semantic of the descriptions with respect to prerequisites and requirements.

Entities such as fine-grained functions, have to offer a suitable and well-defined description of their capabilities, while entities such as agents or engineering tools must comprise compliant descriptions of their needs. The content of such descriptions must be defined in ontologies, and the assignment process is finally done using dedicated matching algorithms. In particular, the matching defines whether given patterns match corresponding templates under consideration of the definitions given in an ontol-

ogy. Such facility shows an essential innovation in this field - automated and semantic-based mapping is the basis for upcoming automation solutions as described in this section 5.4, but can also improve the function search processes as described for PABADIS.

5.5. Concluding Remarks

The main focus of this chapter was on plug-and-participate in plant automation, especially two levels of plant automation have been investigated: MES-level solutions and field-level concepts. For MES-level solutions, state-of-the-art planning systems and their shortcomings were the starting point for a distributed MES layer developed in the PABADIS project. This concept mainly relies on agent technologies for the vertical integration of the automation levels, while the horizontal integration, based on plug-and-participate facilities, provides the basis for distributed planning. Field-level proposals, in contrast, cover three different aspects - the provision of particular and **fine-grained functions** to a flexible infrastructure by **plug-and-participate** concepts. Such functions are identified as the basis for control applications **combining these fine grained functions** as illustrated for IDA and PROFInet, but also as described in the CODING proposal. This function combination finally requires a **dedicated semantic** - the semantic fieldbus approach was introduced. Although this latter mentioned concept is designed for fieldbus systems, it can be easily adapted to general solutions as already described at the bottom of the previous section.

All these concepts - PABADIS, totally distributed automation, enhancements for IDA and PROFInet and finally the semantic fieldbus approach - cover two facets out of three of this thesis: plug-and-participate and plant automation. Both facets have been integrated within this chapter to revolutionary solutions - in a more general view all solutions together show a certain integration of the automation levels: The field-level concepts are based upon each other and will result in a flexible, efficient and robust control solution/architecture for automation control applications, which are finally provided in spontaneous manner to MES-related concepts like PABADIS.

The major contribution of this thesis to the automation field can be concluded in the following way:

- The author deeply evaluated the use of plug-and-participate in the (rather conservative) field of automation.
- One result of this evaluation is the identification and classification of general plug-and-participate requirements in the automation field.
- Based on this information, a MES-level related concept has been developed; its core is an interface, which aggregates these requirements. This concept, moreover, is the crucial point for ensuring the horizontal integration including a homogeneous view to the automation functions, a generic access and a robust and highly flexible infrastructure as the basis for next generation planning concepts (as proposed by PABADIS).
- The concept has been evaluated in two reference implementations based on two classical plug-and-participate technologies.
- Using the Jini-based reference implementation, the concept has been demonstrated by its application in PABADIS CMUs and particular components.

- Finally, field level related plug-and-participate based concepts have been evaluated, which are the basis for integrated and likewise more efficient and robust solutions. These concepts are more or less proposals that must be the subject of further research in order to finally provide innovative solutions in the respective fields.

All these concepts must be investigated with respect to the third facet of this thesis - limited devices. This topic was tackled first when evaluating state-of-the-art automation devices and limited devices in general. It has been shown that legacy automation devices in general can be categorized as limited devices providing limited-device Java platforms. The evaluation finally led to the KVM/CLDC platform as the chosen target platform for the covered automation solutions. Coping with the limited-device character of target devices, all solutions of this chapter are designed with respect to such limited facilities and resources. In PABADIS, for example, this special character of its target platforms is remembered in all design decisions, likewise in the plug-and-participate module: It has been explicitly stated several times that the solution must be limited device suitable. This resulted in a solution avoiding dynamic remote class loading, and requiring only simple serialization and deserialization mechanisms (due to the use of `XMLObject` instance). Finally, a limited-device-suitable plug-and-participate platform is required in order to apply the PABADIS approach on its target platforms - Pini is the resulting plug-and-participate framework, which is described in the following chapter.

CHAPTER 6

Pini - A Plug-and-Participate Technology for Limited Devices

After the evaluation of the requirements of classical plug-and-participate technologies in chapter 2, and the investigation of state-of-the-art automation devices in chapter 3 that can be characterized as limited devices - unable to host classical plug-and-participate facilities - a technology will be presented in this sixth chapter that copes with those limited device platforms. It is called Pini, and its target platform is the KVM/CLDC - the chosen reference platform for automation solutions. This chapter therefore covers the third facet of this thesis, namely limited devices. Strictly speaking, it covers the plug-and-participate facet in conjunction with limited devices. Both together are finally the basis for the already presented automation solutions - in chapter 7 the PABADIS plug-and-participate facility will be migrated to Pini.

Pini, as a result of the PABADIS research of the thesis' author, relies on the abstract plug-and-participate interface, as well as on the plug-and-participate-related abstract automation interface. It can therefore be used for implementing the latter mentioned interface, and thus shows the integration of all three facets of this thesis. In particular, all concepts shown so far are designed with respect to the limited device character of legacy automation devices, and Pini finally allows such concepts to be applied to legacy automation systems.

This chapter mainly focuses on the Pini architecture and its features in order to provide a detailed view of the Pini concept and its innovations, while its relation to plant automation is shown in chapter 7. Such a detailed view of Pini comprises an overview section illustrating the main facets of the concept, as well as a description of the particular implementation of the concepts, and likewise a first evaluation of its resource consumption.

6.1 Overview of Pini

Plug-and-participate technologies are basically motivated by the necessity to alleviate the setup of network infrastructures of devices, or more abstractly of service infrastructures. This fundamental objective has been evaluated in section 2.3.1, resulting in an abstract plug-and-participate interface. According to this interface, concepts must be provided such as community join, dedicated robustness mechanisms, search facilities for specific resources/entities, use patterns for entities, and the description of entities via attributes. Pini as a specific plug-and-participate technology must implement this interface so that the general facilities of plug-and-participate are provided.

In addition to these features defined by the interface, further requirements and constraints have to be considered in Pini's design. For example, it must be suitable for automation systems, and must therefore be compliant to the plug-and-participate-related abstract automation interface as defined in section 5.3.3.2. From this automation relation resulting from its intended field of application, the intended target devices can be derived, namely limited devices of the automation field. This finally leads to Pini's main objectives:

- Pini must be suitable for use on limited devices, especially automation devices.
- Pini must provide reliable and robust infrastructures resulting from spontaneous networking facilities, e.g., single point-of-failure components must be avoided.
- Pini must allow an exact determination of required resources, especially memory. Moreover, it must be as resource efficient as possible in terms of memory and communication load.
- Pini must run on all hardware platforms and communication technologies.

Each of these objectives fundamentally impacts the design decisions of Pini. For example, in order to be suitable for limited devices, Pini's reference platform is the KVM/CLDC. This platform is simultaneously the elected reference platform for legacy automation devices as defined in 3.3.1 and 3.3.2. This also covers the objective of being platform independent - using Java ensures this platform independency.

The most fundamental design decision, however, is with respect to the Pini architecture and its specific interfaces. Pini relies on specific Jini core interfaces and several interfaces from the Jini extension package. It is therefore rather transparent to users whether they use Jini or Pini as the underlying plug-and-participate technology. This kind of transparency and the alleviation of the migration from Jini to Pini (and, of course, vice-versa) is one major reason for this design decision. For example, Jini-based PABADIS CMUs can easily be migrated to Pini as it will be shown in section 7.2. The design decision is further based on Jini's compliance to the plug-and-participate related abstract automation interface. Jini furthermore implements the general abstract plug-and-participate interface, and thus Pini, too. This ensures a powerful Pini platform, which is easily comparable with one classical technology. Such an alleviated comparability shows a second major reason for this design decision.

However, because of Pini's compliance with the Jini interfaces, one might easily consider Pini to be a limited Jini variant - this claim can be invalidated by comparing both technologies. Pini allows the exact determination of required resources and, most inportantly, Pini has a completely different basic architecture than Jini: Instead of a central LUS instance, Pini relies on a hybrid management facility that is described in the following sections in more detail.

A further difference covers the provision and representation of services within the community: Instead of service proxies, Pini uses `ServiceDescription` objects that allow the description of service facilities, and are furthermore the basis for establishing a connection between a service and its clients. Likewise, Pini's attribute handling differs from Jini's counterpart - it allows more capable mechanisms like partial matching, range matching as well as exact matching.

Each of these concepts will be described in the following sections starting with a survey of Pini components and its general architecture and followed by a Pini service and client example. A closer view of Pini protocols and mechanisms is finally the topic of section 6.2.

6.1.1 Pini Components and Features

Pini as an arbitrary plug-and-participate technology generally relies on standard components, namely

- services offering the functionalities to the community
- service users which access the services using the underlying plug-and-participate facility

Both components use dedicated Pini protocols to form spontaneous communities of entities. This behavior requires specific management facilities that support such ad-hoc networking.

Remark: In Jini, a LUS exists as a central instance for management reasons, while UPnP utilizes periodic advertisements in order to maintain the community up to date. These two specific examples show the extreme cases of organizing ad-hoc networks, and therefore show rather orthogonal advantages and disadvantages: single-point-of-failures and disproportional network load.

With respect to the above Remark, Pini uses a hybrid approach consisting of centralized instances, providing a fully-equipped LUS, and decentralized instances, providing only dedicated mechanisms. This hybrid architecture is called the distributed Pini-LUS, and represents Pini's management facility - this is the third component of the Pini architecture.

Considering these components and the abstract interface definitions - especially the necessity to join the community - the basic behavior of Pini components can be determined: They must contact the distributed Pini-LUS in order to either offer a service, or to request for services of interest. This obviously relies on the knowledge of such a distributed Pini-LUS. LUSs in general, however, do not necessarily need to be known in advance. Pini therefore defines a discovery protocol suite, which basically has the same meaning as in Jini.

Remark: The Pini discovery protocols rely on broadcast communication in contrast to Jini's multicast-based discovery protocols.

When a reference to a distributed Pini-LUS is discovered, services use the (Pini) join protocol to register with the commmunity, while clients usually request for services of interest. The requested LUSs respond on client requests with the `ServiceDescriptions` of any services found. These `ServiceDescription` objects are uploaded to the LUS architecture by services during the join process. Such

objects are the essential basis for clients in order to connect to the services for use. In particular, it is used by the Pini environment for providing clients with a service reference object, which forms the basis for service use (more details can be found in section 6.2).

The general behavior of the components reflects the basic Pini architecture, which is illustrated in the following figure:

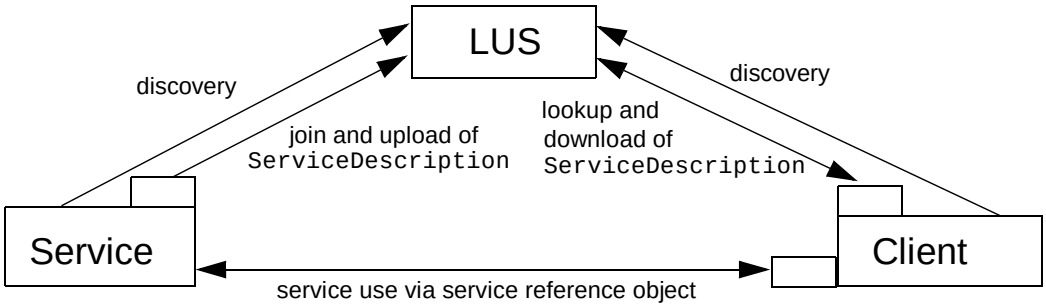


FIGURE 19. The basic behavior of Pini entities illustrating the basic Pini architecture

Remark: This figure is similar to the Jini counterpart, but remote class loading is avoided and the HTTP server is unnecessary. Moreover, the distributed Pini-LUS is aggregated to a simple component in the figure although it basically comprises a distributed architecture: For Pini entities, the distribution of LUSs is rather transparent - they connect a LUS instance and cannot directly distinguish between “full” or “limited” LUSs. More detailed information is given in section 6.2.

The features of Pini sketched here cover only three facilities defined in the abstract plug-and-participate interface of section 2.3.1, namely the spontaneous networking, the search for services and the use of such services. However, the remaining features such as the robustness mechanisms, passive information system and attribute facilities are also covered by Pini: Services can provide attributes describing the features of their functionalities. For this purpose, Pini defines a specific, but generic format - `XMLEntry` objects containing appropriate `XMLObject` attributes. This ensures the provision of flexible description facilities and also allows for powerful and efficient matching.

Pini furthermore defines a remote event facility similar to Jini’s counterpart; i.e., interested parties have to register with distributed Pini-LUSs by specifying the event type, services of interest and a listener reference where an entity listens for incoming events.

Finally, robustness mechanisms are implemented in a lease protocol, in which each registration is annotated with a dedicated lease duration. As in Jini, this lease must be renewed periodically in order to maintain the registration.

These features are described in more detail in section 6.2. Examples of using Pini protocols and facilities are given in section 6.1.3. - they will deepen the reader’s understanding of the Pini architecture by showing a simple Pini service and client. Beforehand, however, the next section will compare the features of Jini and Pini.

6.1.2 Pini versus Jini

Although Jini and Pini are completely different solutions with different fields of application, their features can easily be compared. This results from their common basis, namely the abstract plug-and-participate interface, and finally from the fact that both rely on the same interface suite:

Features	Jini	Pini
management facility	central LUS instance	distributed Pini-LUS resulting from a hybrid approach using centralized instances and dedicated decentralized facilities

discovery	• three discovery protocols:
	  • unicast discovery
	  • multicast discovery
	  • multicast announcement
	• applicable in multicast-enabled networks only
	• result: LUS proxy
		• three discovery protocols:
		  • unicast discovery
		  • broadcast discovery
		  • broadcast announcement
		• applicable in automation communication systems
		• result: LUS reference object

join	• simple registration with LUSs • upload of a service proxy object	• simple registration with LUSs • provision of a `ServiceDescription` object

lookup	• simply requesting the LUS and issuing a description of services of interest via service types (Java data types) and attributes
	• requires remote class loading to provide the proxy object
	• result:
	  • service proxy
	  • attributes
		• simply requesting the LUS and issuing a description of services of interest via service types (Java data types) and attributes
		• does not need any remote class loading for service reference object provision
		• result:
		  • `ServiceDescription`,
		  • service reference object
		  • attributes

remote events	supported, RMI-based, requires remote class loading	supported, no remote class loading necessary
leasing	supported	supported

TABLE 7. The comparison of Jini and Pini features

Features	Jini	Pini
attributes	• arbitrary Java objects having supertype `net.jini.core.entry.Entry` • allows only exact matching on LUS side	• XML tag-based trees are the basis for attributes • attributes are defined by `XMLEntry` objects comprising XML tag-based trees • efficient matching supported, allowing range matching, partial matching, exact matching
transaction	supported	not supported

TABLE 7. The comparison of Jini and Pini features

This table obviously shows that Pini provides nearly the same facilities than Jini, and thus cannot be interpreted as a “limited” Jini, nor as an “implementation-variant” of Jini - the Pini architecture relies on different concepts than Jini, such as the distributed Pini-LUS and `ServiceDescriptions` used for service registration, service use and service search, etc. In particular, these concepts describe Pini as a **completely different architectural approach** than Jini. This **new architecture**, which nevertheless relies on the adopted Jini interfaces, is suitable for limited devices.

An example of basic Pini service and client behavior, given in the following section, shows the similarities between Jini and Pini, and therefore will explain the fact that Pini is almost as powerful as Jini, while section 6.2 will describe Pini’s different architecture despite their similar interface suite.

6.1.3 A Pini Client and a Pini Service

Jini and Pini rely on similar interface suites, so that their applications show essential similarities - this is reflected in an alleviated migration from one technology to the other. This client/service example underlines this statement. It furthermore illustrates the migration of the Jini-based examples given in sections 2.3.2.2 and 2.3.2.3 to a Pini-compliant application.

The first step in developing a Pini application refers to the initialization of the discovery protocols via the (Pini) `LookupDiscovery` class and the instantiation of a (Pini) `DiscoveryListener`:

```
LookupDiscovery lookupDiscovery = new LookupDiscovery(new String[]{""});
lookupDiscovery.addDiscoveryListener(new MyDiscoveryListener(this));
```

CODE EXAMPLE 18. Initialization of the broadcast discovery facility

Compared with code example 2, no difference to the Jini counterpart exists, although different protocols are initialized.

When the `DiscoveryListener` instance returns a reference of a distributed Pini-LUS, services then must register with this LUS as the second step of building a Pini application. This registration, namely

the invocation of the `register(..)` method of the LUS reference, requires several parameters that are similar to their Jini counterparts: A `ServiceItem` object must be created, which internally extracts the `ServiceDescription` object of the service; i.e., the (Pini) `ServiceItem` constructor expects:

- a `ServiceID` (if existing),
- a reference to the service itself instead of a proxy and
- service attributes.

This modified constructor invocation is shown in the following code example, while the modified signature is shown in code example 31:

```
ServiceItem item = new ServiceItem(null, service, null);
```

CODE EXAMPLE 19. Creation of the `ServiceItem` object

The `ServiceDescription` is extracted from the service object - this process is transparent for the developer (see also code example 31) - and the resulting `ServiceItem` object is transmitted to the distributed LUS together with the requested lease duration (60,000 msec) as the second parameter of the `register(..)` method:

```
ServiceRegistration sreg = lusRef.register(item, 60000);
```

CODE EXAMPLE 20. The Pini service registration request

The registration process finalizes in a `ServiceRegistration` object as an acknowledgement of the registration. The `Lease` object is extracted and passed to the lease handling component.

Remark: The lease handling will not be shown here, because it is simply a `Thread` running parallel to all other actions and thus simply invokes the `renew(..)` method of the leases.

The client application uses Pini-LUS references in order to gain an overview of the community - they provide a search pattern to the service search facility defined by the abstract plug-and-participate interface. Pini implements this facility in its `lookup(..)` methods, which expect a `ServiceTemplate` object. This object comprises the same information as its Jini counterpart:

```
ServiceTemplate template = new ServiceTemplate(null,
                    new Class[]{Class.forName(IRoutingService), null});
```

CODE EXAMPLE 21. `ServiceTemplate` creation

The lookup process itself is rather simple, namely the invocation of the `lookup(..)` method of the LUS reference:

```
ServiceDescription o = reg.lookup(template);
```

CODE EXAMPLE 22. A lookup request

Here, a fundamental difference to Jini does exist: While Jini directly returns the service proxy object as the response of a lookup request, in Pini clients instead receive the particular `ServiceDescription` object, which is then the basis for accessing the service - if the `getServiceReferenceObject()` method is invoked, the initialized reference is provided and can be used for accessing the service functionality. For example, in this case an `IRoutingService` object is retrieved, and a route can be requested as illustrated in the following code example:

```
IRoutingService rs = (IRoutingService) o.getServiceReferenceObject();
Route r = rs.getRoute(..);
```

CODE EXAMPLE 23. Retrieval of a service reference object and service use

These basic facilities demonstrate the already explained similarities of Pini and Jini, although fundamental details in the interfaces differ. It also shows that Pini is nearly as powerful as Jini. This will be further underlined by the participation of the Pini client in the passive information system: Striving to be always up to date, the client also registers for remote events. In other words, it prepares:

- a `ServiceTemplate` object defining the services of interest
- the listener instance, which receives events occuring in the community
- the event type describing whether services of interest are joining or leaving the community or simply modifying their attributes.

While the `ServiceTemplate` and the event type do not essentially differ from Jini's mechanism, the listener instance does rely on different facilities than Jini, especially in order to avoid Jini's dependency on Java RMI. Pini therefore provides a specific class, namely the `PiniRemoteEventListener`, which offers all required basic features for the Pini remote event notification architecture. This class is to be extended by a concrete listener instance, as code example 24 shows:

```
class MyRemoteEventListener extends PiniRemoteEventListener{
    public MyRemoteEventListener(){
      super();
    }

    public void notify(RemoteEvent e)throws RemoteException{
       . . .
    }
}
```

CODE EXAMPLE 24. The remote event listener class of that client example

After all parameters are prepared, clients can register for events by invoking the `notify(..)` method of the Pini-LUS reference. This method, however, has a different signature than its Jini counterpart - it concentrates on necessary information only. For example, the handback object has been excluded from the architecture, while the most fundamental modification concerns the listener instance: Instead of a complete listener instance, only a description is sent to the LUS in order to save resources on the LUS side, and, most important, to avoid remote class loading. Details on this Pini remote event facility can also be found in section 6.2.

This finally results in a modified method invocation, which can be seen in the following code example:

```
//register for remote events
EventRegistration ereg = reg.notify(template, transition,
                         listener.getDescription(), 60000);
```

CODE EXAMPLE 25. Registration for remote events

The received `EventRegistration` object mainly consists of the assigned lease, which is passed to the lease handling component.

This example has shown differences between Pini and Jini in their interfaces - this can be explained by their different architectures. However, the easiness of migrating from Jini to Pini and vice-versa can be validated. A further example in this respect is given in section 7.2, where a Jini-based PABADIS CMU is migrated to Pini as the underlying plug-and-participate technology. This will finally validate the design decision to rely on an interface suite similar to Jini. The interface suite is furthermore the basis for the following section on Pini facilities - it mainly shows Pini as a different architecture than Jini.

6.2 Detailed Realization of Pini Facilities

The overview of Pini provided a description of its general architecture, sketched major design decisions and also surveyed resulting components and the relationships among them. The Pini architec-

ture has been further illustrated by an example of a service and its corresponding client. However, no particular details on the mechanisms and protocols used have been given so far. This required detailed look - given in this section - basically covers facilities defined by the abstract plug-and-participate interface. Further aspects to be considered in the development of the general plug-and-participate features are the basic requirements of these common plug-and-participate concepts. Both parts therefore define the obvious structure of this section: First, a description of such fundamental Pini features is given as the basis for the second part, which focuses on general plug-and-participate mechanisms as defined by the abstract plug-and-participate interface from section 2.3.1. Basic Pini features, for example, are an abstraction of network facilities, a dedicated `PiniRMI` framework in conjunction with a specific serialization mechanism as well as essential features for services and clients aggreated in the `DefaultPiniService` and `DefaultPiniClient` classes, respectively. Also appropriate mechanisms for community management are tackled allowing the efficient setup of Pini service infrastructures. The most fundamental feature of the Pini plug-and-participate technology is the already mentioned `ServiceDescription` mechanism, which closes this section on basic facilities. Based on these essential concepts, the general plug-and-participate implementation is shown, and structured with respect to the abstract interface from section 2.3.1.

6.2.1 Basic Pini Prerequisites

Basic Pini features are the fundament of Pini's ability to provide a powerful plug-and-participate technology for limited devices. Concepts like the network abstraction layer and the `ServiceDescription` facility are the basis for those features defined in the general plug-and-participate interface.

6.2.1.1 Pini Network Abstraction

One of Pini's objectives is the provision of a technology that is completely independent of any underlying (Java) platform. Therefore, Pini strongly relies on standard Java classes/features in order to be platform independent. The KVM/CLDC as Pini's reference platform, however, provides essential features only as extension packages, but not as standardized functions. This leads to the definition of suitable abstraction layers, for example, for network communication: A second aspect in addition to the platform independency is Pini's independency of any underlying communication technology - Pini must be suitable for relevant industrial technologies such as fieldbuses, bluetooth, or even combinations of them (as shown in [8]). It must be furthermore applicable to common IP-enabled networks based on Ethernet or similar technologies. Pini's network abstraction, therefore, basically provides mechanisms for all required communication features such as broadcast communication, stream-based and packet-based communication as well as server socket facilities. Such requirements are reflected by the abstraction layer by providing the following classes and interfaces:

- `NetworkHandler`: This class provides a simple reference to arbitrary `NetworkConnector` instances, which are the central facility of the network package.
- `NetworkConnector`: This data type allows the Pini components to retrieve arbitrary datagram connection objects (`UnicastDatagramConnection`), stream-based connections (`UnicastStreamConnection`), broadcast facilities (`BroadcastConnection`) and `ServerSocket` instances.
- `UnicastDatagramConnection`: This interface simply provides datagram-based communication mechanisms for Pini components.

- `UnicastStreamConnection`: This interface abstracts arbitrary stream-based communication facilities, and thus provides input and output streams for Pini components as usual in Java.
- `BroadcastConnection`: This interface offers broadcast communication facilities to Pini components.
- `ServerSocket`: This type simply provides an abstraction of arbitrary `ServerSocket` mechanisms.
- `Connection`: An object representing a connection. It is usually returned by a `ServerSocket` instance, which received a connection.

The interfaces `BroadcastConnection`, `UnicastStreamConnection`, `UnicastDatagramConnection` and `ServerSocket` provide the abstraction: They must be implemented in a customized fashion for Pini and the particular environment. Pini is therefore completely independent of their implementation, because Pini components transparently access network facilities through these interfaces. For example, the Pini discovery protocol essentially relies on the `BroadcastConnection` interface, while other Pini components mainly use unicast connections and server sockets abstracted by their respective interfaces. A further Pini communication basis is provided by the `PiniRMI` concept, which is defined on top of the network abstraction. This framework is topic of the following section:

6.2.1.2 `PiniRMI` and Pini Serialization

The general intention of `PiniRMI` is the provision of an object-oriented concept for the seamless invocation of remote methods. This facility needs to be as resource-efficient as possible. The `PiniRMI` concept, therefore, concentrates on general remote method invocation functionality: Following the object-oriented programming paradigm, a Pini object retrieves a reference to a remote object usually as the response of an interaction, such as a Pini service search request.

Remark: General RMI concepts often provide a naming service facility, which can be used to retrieve object references - `PiniRMI` is customized to Pini requirements and does not need such a feature.

A Pini remote object reference is represented by a stub implementing the interface(s) of a specific object, while on the remote object end a skeleton is situated that receives incoming invocation requests. Such requests are finally forwarded to the addressed object. This stub-skeleton mechanism of `PiniRMI` encapsulates a dedicated RPC facility as described in [28] - it is the main basis for the efficient remote method invocation, while its circumjacent stub-skeleton concept keeps the object-orientation. Stub and skeleton classes are generated by a special compiler resulting from the author's work in this field.

Remark: This compiler does not run on limited devices - the development of Pini applications is usually done on general development platforms running on arbitrary PCs. The compiler uses Java reflection mechanisms for class introspection so that stub and skeleton classes can be generated appropriately.

The encapsulated RPC mechanisms rely on a private protocol, which is customized for Pini and its communication requirements - it ensures resource-saving mechanisms for Pini, and essentially ensures Pini's applicability to limited devices. The protocol is rather simple, because it only relies on client - server relationships: A client sends a request to a server, which interprets the call, performs the invocation and returns the result. Within this relationship, a client is represented by a stub. It initializes the `Communicator` class and provides the target address and port. The method invocation is simply a

call to the `callMethod(..)` method of the `Communicator` instance. It expects the name of the method to be called and accompanied parameters are provided. Calling the method transmits the request to the server represented by its respective skeleton.

The invocation parameters are encapsulated in an object array - an essential part of a request transmission is the handling of such parameters, which must be serialized by a special Pini serialization mechanism described later in this section. The transmission of requests is furthermore related to the network abstraction. For example, the `Communicator` class and the server-related mechanisms simply use the facilities provided by the network abstraction. On the server side, requests are received by a `MethodCallListener` instance using the `ServerSocket` facility of the network abstraction. After de-serialization, the request is queued until a `MethodCallEvent` is generated by a dedicated `ParserThread`. This event is immediately passed to a `MethodCallEventHandler` instance, which invokes the method specified - a MethodCallEventHandler is represented by the skeleton of the affected server object. If the method invocation has finished, the result is sent back via a specific `ResultDelivery` component, even if a `void`-method has been invoked. Both, the invocation request and the result transmission, use unicast connection facilities provided by the network abstraction, namely the `UnicastStreamConnection` mechanism.

This basic `PiniRMI` architecture is illustrated in the following figure in order to explain its structure, as well as to emphasize its simplicity:

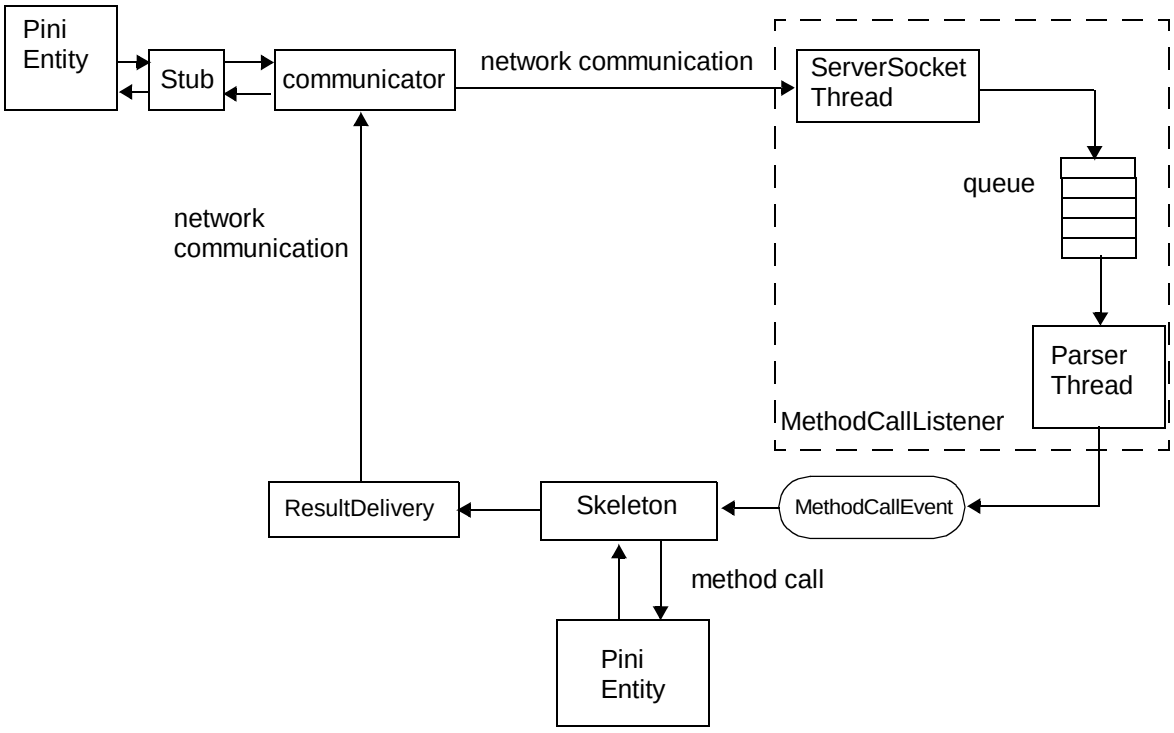


FIGURE 20. The basic structure of a `PiniRMI` method invocation

While on the client side the concept is rather simple, since the client only retrieves the stub object as a result of an action and does not need to install any underlying `PiniRMI` instances, the server side is more complex - i.e., a `MethodCallListener` and the skeleton must be initialized. This complex structure must be hidden from the user/developer. In particular, the structure is transparently aggregated in the `PiniRemoteObject` class, which must be extended by each object providing remote facilities. It automatically initializes the required infrastructure and facilities transparently for the user, but shows a fundamental advantage: Each Pini entity runs only a single instance of a `MethodCallListener`, regardless of the number of skeletons. Such a dedicated instance uses a hashtable allowing a mapping between requests and addressed skeletons. The hashtable contains IDs as keys for the stored skeletons. This fundamental architecture design leads to a specific initialization procedure of the underlying `PiniRMI` infrastructure: If a `PiniRemoteObject` is initialized, it checks whether a `MethodCallListener` is already installed. If not, an instance is created, and the initial skeleton is stored in the hashtable. Otherwise, if a listener is already installed, the skeleton object is simply added to the existing table. Each skeleton is therefore annotated with a specific ID, which must be passed to each corresponding stub. Incoming method call requests are then dispatched to the addressed skeleton identified by the ID. This ID is extracted from the invocation request, since the parameter transmitted first is interpreted as the ID of the corresponding skeleton.

Running only a single `MethodCallListener` for all local skeletons is reasonable - it conserves resources and is an essential optimization with respect to limited devices. If each skeleton would run a separate listener, a corresponding number of `Thread` and `ServerSocket` instances must also be maintained, which is rather resource consuming, as they are waiting most of the time.

A further resource consumer in terms of network communication and remote method invocation is the transmission of objects (such as invocation parameters) in an interpretable fashion. Such objects must be serialized at the client in an efficient way, so that they can easily be de-serialized on the receiving end. `PiniRMI` therefore defines its own serializer module. Defining such a proprietary and customized module is motivated by the required efficiency on the one hand, while a second motivation, on the other hand, is the simple fact that limited device platforms like KVM/CLDC do not provide such facility due to its potentially enormous resource consumption. Pini's serialization mechanism is therefore rather resource efficient, and it simply concentrates on the interpretation of object data rather than on the retrieval of data from objects: While general serialization mechanisms devote much effort to the introspection and serialization of objects, Pini retrieves the data from an object via dedicated methods; i.e., each Pini-serializable object must implement the `PiniSerializable` interface:

```
public interface PiniSerializable{
  Object[] getFieldValues();
  void setFieldValues(Object[] fieldValues);
}
```

CODE EXAMPLE 26. The `PiniSerializable` interface

Objects, that must be transmitted through the network, such as method invocation parameters, must implement this interface. Developers must therefore specify the data of the object to be transmitted - such data must also be `PiniSerializable`. For example, the `ServiceItem` class is `PiniSerializable`, because it is sent to the LUSs via an `PiniRMI` method call. The data contained in a `Ser-`

viceItem are the ServiceID, the ServiceDescription and the attributes - developers must handle the data objects in the aforementioned methods, while the objects themselves must be serializable as well.

The serializer module is rather simple and contains only three methods as shown in the following code example:

```
public final class Serializer{
  public static void writeObject(DataOutputStream out, Object dataObject)
                                                    throws IOException{…}
  public static Object readObject(DataInputStream dis)throws Exception{…}
  public static void toArray(Object destination, Object source){…}
}
```

CODE EXAMPLE 27. The interface of the Serializer class

Likewise, the serialization process is rather simple: If an object needs to be serialized, the write(..) method is invoked, which expects the object to be serialized and the stream to which to write the extracted bytes. The serialization process, in turn, is a simple recursion; i.e., first the class name of the object is determined and written to the stream. The next step determines the hashcode of the object, and this is also written to the stream to avoid cyclic serializations.

Remark: Objects can refer objects that have references back to the original object - such cycles must be resolved so that already serialized objects are not serialized again. In such case, the hashcode is written to the stream, so that during de-serialization the respective object can be used to restore the cyclic references.

In the third serialization step, the fields of the object are obtained, if the getFieldValues() method is invoked. The number of fields is written to the stream, and finally the objects are serialized by recursively invoking the above mechanism. However, the algorithm is only invoked recursively if the fields are not of primitive data types that can easily be serialized. If the recursion stops due to a primitive

data type, the algorithm determines the particular type and writes the type name and the value itself to the stream. This results in a rather simple structure of serialized objects as illustrated in figure 21:

```
Class name [a]
Hashcode [code]
Number of fields [k]
   Class name [a1]
   Hashcode [code1]
   Number of fields [n]
      Class name [primitive value' name]
      Value [primitve value - end of recursion]
   …
```

FIGURE 21. The general structure of a serialized object

The de-serialization process, initialized by the `readObject(..)` method, works in a similar fashion: It reads the class name and the hashcode from the stream and checks whether this object is already known. If so, the de-serialization is finished, and the already de-serialized object is returned. Otherwise, the number of fields is read, and subsequently the fields are taken from the input stream in a similar recursion as depicted for the serialization process. The recursion stops if there are no further values/fields to be de-serialized. When all fields are restored, they are subsequently passed to the object to be de-serialized via the `setFieldValues(..)` method, and the de-serialized object is finally returned.

Both facets together define a rather efficient and resource-saving algorithm for object transmission through the network, which is a major fundament of the `PiniRMI` concept - it ensures the transparent submission of method invocation parameters among objects as a basis for transparent networking. This serialization facility and the stub-skeleton mechanism are an important basis for making `PiniRMI` suitable for limited devices, especially for automation platforms. Moreover, `PiniRMI` relies on the network abstraction layer described in the previous section, and thus can be applied to different environments such as automation-related networking or Internet-like infrastructures. This finally is one fundament for Pini's efficient plug-and-participate concept and its ability to run on different network technologies transparently. For example, the `PiniRMI` framework is a major basis for the service provision and of general service features, which will be described in section 6.2.1.4. `PiniRMI` is furthermore the main fundament for Pini's service use pattern - namely the concept of service proxy stubs, which is given in section 6.2.2.3.

6.2.1.3 Community Management

Community management of plug-and-participate networks is an integral feature that is provided in different ways by different technologies: While in UPnP, the network management is implicitly provided by the advertisement mechanism, Jini provides the community management explicitly with a dedicated component called Lookup Service. Pini also uses such a dedicated component, but in contrast to Jini, a distributed LUS architecture is defined that avoids the risk of single point-of-failures present in centralized components (see also section 6.3) as well as the disproportional network load (and redundancy) caused by the UPnP architecture.

The distributed Pini-LUS architecture defines both designated LUS components that work in a rather centralized fashion and it furthermore comprises decentralized facilities. Centralized LUS components participate in all mechanisms specified by the abstract plug-and-participate interface such as join, robustness, service search and a passive information system, while decentralized modules do not participate in the passive information system. These decentralized modules are incorporated into each Pini service so that each service is always able to join a community. Hence, services can provide their functionality to the network in all cases, even if no central LUS is reachable. Clients, in turn, can always request the community for services, even if no central LUS can be discoverd. This ensures reliable systems and service - service user relationships, since clients can obtain LUS references if services are in the network. These LUS references can then be used to search for services of interest. However, decentralized LUS modules are limited to local services only - remote services are rejected if they want to register. Without this constraint, resources might be exhausted, and the system load increases disproportionally due to the unnecessary redundancy and additional network communication. This also impacts the robustness mechanisms, which only need to deal with local service registrations instead of community-wide robustness and maintenance. The service search functionality, in contrast, does not differ from that provided by the designated central LUS components.

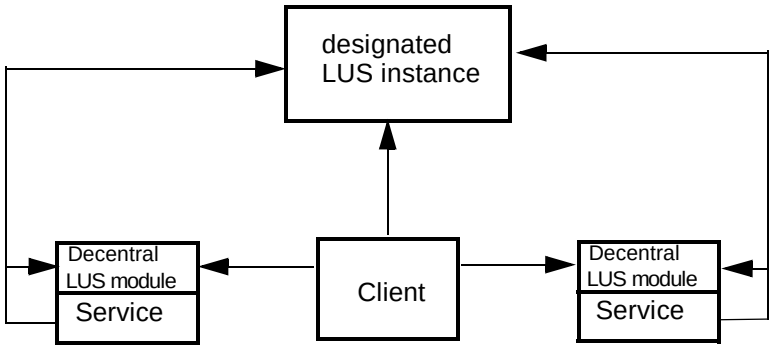


FIGURE 22. The architecture of the distributed Pini-LUS

Figure 22 shows the general architecture of the distributed Pini-LUS: While services provide their own dedicated management modules, clients do not - they simply participate in the community without differentiating among obtained LUS references. Moreover, services transparently join the community and cannot distinguish between their own local LUS instance and global LUS instances.

This distributed Pini-LUS as the community management system combines the advantages of completely decentralized architectures and those using a central instance - an innovation in this field. It ensures a high reliability of the system, and shows rather modest resource requirements. The distributed Pini-LUS relies on a peer-to-peer-like infrastructure and is a result of the author's research in the field of service provision and service search. A further impact on this architecture results from the investigation of the community management facilities of classical plug-and-participate systems.

Such an explicit community management like that of the distributed Pini-LUS requires particular mechanisms/protocols for Pini entities to participate in a community without detailed knowledge of the infra-

structure. Strictly speaking, Pini entities must be able to obtain connections/references to the LUS instances during runtime instead of having to specify LUS locations in advance. Pini, therefore, defines a discovery protocol suite that is similar to Jini's counterpart, but concerns itself more with automation needs and Pini's field of application, namely limited device platforms. It likewise specifies three subprotocols having similar objectives to Jini, but with different prerequisites. Instead of multicast communication, the Pini discovery protocol suite uses broadcast communication, although broadcast shows at least two shortcomings when compared with multicast:

- Broadcast, in general, causes a higher network load than multicast communication - it generally floods the network with the message data.
- Broadcast has a smaller range than multicast. While broadcast is usually limited to a single subnet or even a domain, multicast is normally available throughout the Internet, if it is allowed/supported by the respective Internet providers.

Remark 1: Depending on the multicast protocol used, broadcast could also be the basis for multicast communication (see [20]).

Remark 2: The limitation to domains/subnets is for administrative reasons but not for technological. This is due to the aforementioned high network load caused by broadcast communication.

However, within the Pini architecture, the use of broadcast is reasonable, because

+ almost all communication technologies for industrial automation support broadcast communication natively, while multicast communication is not necessarily implemented.

An evaluation of this design decision is given in section 6.3, while the following paragraphs mainly concentrate on Pini's discovery protocol suite comprising

- the unicast Pini discovery protocol
- the broadcast Pini discovery protocol
- the broadcast Pini announcement protocol

The objectives of these protocols are similar to the Jini counterparts: The unicast discovery protocol is an active protocol that requires knowledge of the LUS to be discovered. Pini simply provides a unicast connection to the LUS. This protocol is implemented in the `LookupLocator` class and is completely based on the network abstraction. The broadcast discovery protocol is also an active protocol, but in contrast to the unicast discovery protocol, it does not require knowledge of LUSs. It sends several discovery requests, consisting of information regarding LUSs of interest and the IDs of already-known LUSs, via broadcast to the network. The protocol issues seven requests every five seconds, and after finishing this, it switches to a passive mode. In particular, the discoverer switches from the broadcast discovery protocol to the broadcast announcement protocol transparently. This protocol is a passive one and likewise does not require knowledge of LUSs. It is based on the periodic announcement of LUSs sent to the community so that interested entities can track the LUS availability. Via this protocol entities get informed about LUSs joining the community, after the active mode of a "discoverer" has finished. Both protocols are defined in the `LookupDiscovery` class; while the broadcast discovery protocol must be actively started, the use of the announcement protocol is transparent. Of course, parallel to this, the broadcast discovery protocol runs a `Thread` with a `ServerSocket` instance, that is responsible for receiving incoming discovery responses.

All protocols along with the distributed LUS architecture define the relationship between the community management and the discovery facility - each LUS component must participate in all discovery protocols as respective communication partners: They must listen for incoming discovery requests and periodically announce their availability. However, it is transparent for entities whether they discover a designated (central) LUS component or a dedicated LUS module provided by a service. Both mechanisms always result in LUS references having the same interface: the `ServiceRegistrar` interface as shown in the description of Jini in section 2.3.2. Such references are the basis for almost all mechanisms of Pini that represent implementations of facilities defined by the abstract plug-and-participate interface. They are topic of section 6.2.2.

6.2.1.4 `DefaultPiniService` and `DefaultPiniClient` Facilities

The provision of plug-and-participate facilities in Pini requires certain configurations as well as the initialization of dedicated components. For example, services and clients must obtain their addresses and the interfaces they implement (in case of services) or support (in case of clients). Services additionally have to initialize underlying network components such as skeleton, `MethodCallListener`, etc. The Pini environment defines two distinct classes for that purpose, offering these features in a transparent fashion: `DefaultPiniService` and `DefaultPiniClient`.

`DefaultPiniService`

All mechanisms required for proper service behavior are concentrated in the `DefaultPiniService` class - each service must extend this class in order to inherit these features. A service has to install a skeleton object so that clients can properly access the service functionality. This defines each service as a `PiniRMI` remote object, and thus services must also extend the `PiniRemoteObject` class. Services therefore transparently install the required underlying `PiniRMI` architecture, namely the `MethodCallListener` and the accompanied skeleton. This finally provides a Pini service with information regarding the skeleton ID (remember the `PiniRMI` concept), as well as the service port, on which the `MethodCallListener` listens for incoming requests. Both parameters must be published to the community in order to allow clients to properly contact/access the service - these data are stored in the `ServiceDescription` instance.

In addition to this service access functionality, each service must evaluate a configuration class in order to learn about its local system.

Remark: Pini uses a specific but simple class for configuration information. This results from the fact that automation systems, and limited devices in general, do not necessarily define a file system, and thus obtaining the configuration information from a file might be impossible - classes, however, can always be loaded.

Such local system information, for example, comprises the address of the entity and the interfaces this particular service implements. The information is stored (as strings) in the configuration class, because limited device platforms often do not provide mechanisms to obtain such information from the underlying system. The retrieved data will be checked during the evaluation so that a certain robustness and reliability can be ensured. For example, the address is confirmed by creating a connection to this address. The interfaces defined in the configuration information are checked whether the service is of type of the given interfaces. Such restrictive mechanisms are required, since the information is published to the community within `ServiceDescription` instances representing the service and defines

the access path. If a service provides corrupt or invalid information, i.e., a wrong address or invalid interfaces, clients as well as services may run in undefined states - especially in automation systems, this must be avoided. The `DefaultPiniService` class therefore includes dedicated mechanisms for these tests as illustrated in the following code example:

```
try{
    for (int i=0;i<interfaces.length;i++){
    Class localClass = Class.forName(interfaces[i]);
    if (localClass.isInstance(service)){
      //everything is fine
    }else{
      //throw an exception
    }
  }
}catch(Exception e){...}
```

CODE EXAMPLE 28. The evaluation whether a service implements the classes defined in the configuration class

The third service-specific aspect is the initialization of the LUS module - as depicted in the previous section, each service runs its own dedicated LUS module in order to be always able to join the community. This simply involves the generation of the module by invoking its constructor. All components and protocols are started automatically in order to participate in the distributed Pini-LUS architecture.

All these sketched facilities - namely the provision of the underlying service access path via a service skeleton, the evaluation of configuration information and the initialization of the LUS component - are aggregated in the `DefaultPiniService` class, which uses the following interface:

```
public class DefaultPiniService extends PiniRemoteObject{
  public final void init(String path, Object service){…}
  public final String getAddress(){…}
  public final String[] getInterfaces(){…}
  public final int getServicePort(){…}
  public final int getCode(){…}
}
```

CODE EXAMPLE 29. The main methods of the `DefaultPiniService` class

Each service must extend this class, so that it automatically inherits the required facilities. Otherwise, if a service neither extends the class nor invokes the `init(..)` method, exceptions are thrown so that the service will not be published in the Pini community.

`DefaultPiniClient`

In the same way as Pini services, also Pini clients require dedicated configuration information, especially the address information. The setup of an underlying `PiniRMI` infrastructure or the initialization of a LUS module is not necessary. Thus, the functional range of the `DefaultPiniClient` class is efficiently reduced to the evaluation of configuration data - this mainly concerns the validation of the

address information, and optionally also the evaluation of specific interface information. While the address is mandatory, interfaces do not necessarily need to be provided. In particular, the client-related interface information is composed of the dedicated names of so-called proxy stub classes (see also section 6.2.2.3), which are the basis for service access. If such names are specified, a Pini client is limited to exactly these proxy stubs. If, in contrast, no limitation is given, a client can initialize all proxy stubs it finds in its local class path (see also section 6.2.2.3; here the service use is described in a detailed fashion).

Considering these dedicated facilities of Pini clients, the `DefaultPiniClient` class shows a similar but limited interface:

```
public class DefaultPiniClient{
    public final void init(String path, Object client){}
    public final String[] getInterfaces(){…}
    public final String getAddress(){…}
}
```

CODE EXAMPLE 30. The `DefaultPiniClient` class

The information provided by the `DefaultPiniService` and `DefaultPiniClient` class is used by Pini mechanisms, which finally implement the abstract plug-and-participate interface. For example, the addresses as well as the interfaces are the fundament of both service use and service provision - they are both stored within `ServiceDescription` instances published to the Pini community; `ServiceDescription` is topic of the following section:

6.2.1.5 The `ServiceDescription` Facility

Pini services have to register with the community and must publish information about themselves, such as attributes describing the service functionalities, but especially essential information regarding the service access path. Such information alleviating and ensuring an efficient service use is provided by Pini's `ServiceDescription` class. It contains the names of interfaces a service implements, the address of a service as a `String` representation, and finally the port of the `MethodCallListener` instance accompanied with the skeleton object of the service. All data are retrieved from the underlying `DefaultPiniService` class during the join process - exactly at the moment when the `ServiceDescription` instance is created. `ServiceDescription` objects are sent to the distributed Pini-LUS in order to represent the service in the community. Clients, in turn, request the `ServiceDescription` objects from the LUSs in order to access the particular service: They use this information for establishing connections to the service and use the attribute information for choosing among the services found. A `ServiceDescription` is therefore the basic fundament for service access, as it will be described in section 6.2.2.3. Moreover, this facility presented in [28] by the thesis' author and his colleague, has been adopted by another research group from the ETH Zurich in order to provide a technology similar to Pini - they called it Mini (see [43, 44], and also section 6.3). This especially proves our concept with respect to its reasonability and its generic fashion. Moreover, among further aspects, this facility is one major innovation in the field of plug-and-participate and is an essential basis for efficient networking even on limited devices. This, obviously, can increase the added value of such devices, especially of automation devices.

6.2.2 Implementation of the Abstract Plug-and-Participate Interface in Pini

The implementation of the abstract plug-and-participate interface in Pini relies on the basic prerequisites sketched in the previous section, and also on the interface suite that has been adopted from Jini. Pini's plug-and-participate facilities therefore show similar behavior than corresponding Jini features, while they rely on different basic concepts. In this respect, the underlying community management system, the service provision concept as well as dedicated service use patterns must be emphasized - they are the fundamental innovations in Pini that allow the provision of a powerful plug-and-participate concept even for limited devices and especially for automation environments.

The main focus of this section lies in the implementation of the abstract plug-and-participate interface specifying concepts like community creation, service search, robustness, passive information system, attributes and service use patterns - the section is structured respectively.

6.2.2.1 Spontaneous Networking of Pini Entities

The basic aim of plug-and-participate systems is to provide networking facilities for building flexible infrastructures of entities. A second aspect tackles the establishment of service - service user relationships, which finally rely on the first aspect, namely the infrastructure setup.

Pini provides a join protocol that allows services for transparently offer their functionalities to appropriate service infrastructures. Such infrastructures, however, do not necessarily need to be known to services in advance. The Pini framework supports entities in obtaining an overview of the community, especially with respect to the community management system: Using the Pini discovery protocol suite enables entities to receive references of the management system represented by references of the distributed Pini-LUS architecture. In other words, the first step in spontaneous networking is the discovery process, which is always the same for each operation in a Pini system.

With the help of such references, services can offer their particular service functionality - they join the community by means of the Pini join protocol. This join protocol simply defines the upload of service-related information to distributed Pini-LUSs. Such information comprises a unique identification of the service, the description of its access path, and finally attributes describing the service functionalities in a detailed fashion. With respect to the similarity between Jini and Pini (and the alleviated migration from Jini to Pini and vice versa), the data are also encapsulated in `ServiceItem` objects. However, despite the general similarity with Jini's `ServiceItem` class, in Pini such objects expect different information (except the `ServiceID`). The major differences concern the attributes and the references to the service, as shown in the following code example:

```
public ServiceItem(ServiceID serviceID, DefaultPiniService service,
                                                    XMLEntry[] attrSets){
   this.serviceID = serviceID;
   this.attributeSets = attrSets;
   this.serviceDescription = ServiceDescription.getDescription(service);
}//end of constructor
```

CODE EXAMPLE 31. Creation of the `ServiceDescription` object in the `ServiceItem` constructor

While in Jini a service proxy object is expected, Pini requires a reference to a `DefaultPiniService`, which is usually a reference to the service object itself. This reference allows the Pini join protocol to retrieve any necessary information regarding the access path: As pointed out in the example, a `ServiceDescription` instance describing the service access path is created. Moreover, instead of `Entry` objects, the Pini attribute mechanism expects `XMLEntry` objects, which contain appropriate `XMLObject` instances. These objects finally embody the detailed service information (see also section 6.2.2.6). This modifaction does not result in any limitation compared with Jini's freedom to define arbitrary attribute objects having the supertype `Entry` - `XMLObjects` are able to describe almost all kinds of data, and allow for efficient matching.

Such a `ServiceItem` object is finally published to the Pini community by uploading it to distributed Pini-LUSs - invoking the `register(..)` method of a LUS reference finalizes the join process:

```
ServiceRegistration serviceRegistrar = reg.register(item,
                                                    defaultLeaseDuration);
```

CODE EXAMPLE 32. Registration of a Pini service with a Pini-LUS instance

An important aspect in this join process is with respect to the distributed Pini-LUS: For Pini services, it does not matter whether they register with their own local LUS instance or with a central LUS entity. This means, a registration request is simply the invocation of the `register(..)` method of the Pini-LUS, which is acknowledged by a `ServiceRegistration` object as shown in the previous code example. This object provides access to the particular registration. For example, it provides the annotated lease object and also allows operations with respect to the attribute handling facility of Pini. All these interactions also impact the LUSs, which must store the data - the `ServiceRegistration`is a `PiniRMI` remote object extending the `PiniRemoteObject` class. A service only handles a particular stub, while the registration object itself is situated at the LUS. In this way, resources can be conserved on the service side, as a stub usually requires fewer resources than an entire `ServiceRegistration` object.

Considering the mechanisms sketched in this section, the Pini join protocol not only supports services in announcing their availability and dedicated information regarding its functionality; it furthermore provides detailed information for establishing the service access path. This information describing the service in a detailed fashion is the basis for the service search mechanisms, while the service access path description, aggregated in `ServiceDescription` instances, concerns the service use patterns as defined in Pini. Both are described in separate sections.

6.2.2.2 Service Search

The counterpart of the service announcement is the service search facility. It relies on two basic requirements: a reference to the Pini community management system - namely, a reference to a distributed Pini-LUS instance - and a description of services of interest. While a reference to a distributed Pini-LUS is retrieved from the Pini discovery protocol suite, the description of services of interest must be prepared by a Pini client itself. Such a search pattern to be provided for the Pini service search mechanism is defined by a `ServiceTemplate` object (in order to remain compliant with Jini's interface suite). A (Pini) `ServiceTemplate` comprises three particular information categories:

- A `ServiceID` describing exactly one service in the Pini community.
- Service interfaces describing the demanded types of services of interest.
- Service attributes describing the features of demanded services.

While the `ServiceID` and the attributes are directly sent to the LUS, the service interfaces are converted to a string representation of their names. Strings are used for this purpose in order to alleviate the transmission of the search pattern as well as the comparison with registrations stored in the LUS repositories. The Pini `ServiceTemplate`, moreover, expects `XMLEntry` instances in order to allow an efficient matching, which is the major objective of service search: Service registration data stored in the LUS repositories also contain such `XMLEntry` attributes, as well as the search pattern. This, consequently, allows the application of the matching algorithm as it is described in section 5.3.3.3.7, since these `XMLEntry` objects contain simple `XMLObject` attributes - they make up the input for the matching algorithm.

When the search pattern is created, the search request can be issued to the community management system. This means, simply one of the `lookup(..)` methods of a distributed Pini-LUS reference is invoked - similar to Jini, the distributed Pini-LUS also provides two distinct methods for this purpose:

(1) `ServiceDescription sd = lusRef.lookup(template);`

(2) `ServiceMatches matches = lusRef.lookup(template,number);`

Although these methods have rather similar functions to those in Jini, they deliver completely different results. Method (1) returns a `ServiceDescription` object instead of a service proxy, while method (2) returns `ServiceMatches` objects. These `ServiceMatches` also comprise an array of `ServiceItem` instances carrying `ServiceDescription` objects instead of service proxies. Using the obtained `ServiceDescription` objects, clients retrieve the corresponding service reference object simply by invoking the `getServiceReferenceObject()` method (see section 6.2.1.5), which provides the final access path to the service.

Such particular behavior of the Pini service search facility, as well as the specific fashion of providing service access, ensures Pini's compatibility with limited devices - `ServiceDescription` objects do not require any remote class loading, and thus the memory requirements can be determined in advance (see section 6.3 for further details). A `ServiceDescription` finally provides clients with the access path to services, which is topic of the following section.

6.2.2.3 Service Use in Pini

Establishing spontaneous service communities and service search are two basic plug-and-participate aspects - the third lies in an alleviated creation of service - service user relationships. The concept, which is the basis of Pini's service use facility is the third major innovation in plug-and-participate for limited devices - preinstalled service proxy stubs relying on the service interfaces allow the exact determination of the resource consumption in advance, avoid remote class loading and provide a transparent service reference.

In Pini, the first step is the retrieval of `ServiceDescription` objects via the Pini lookup protocol, while the second step concerns the establishment of the connection. In particular, a service reference object must be instantiated using the information from the `ServiceDescription`. This instantiation is a multi-stage process: First, it is determined whether the `ServiceDescription` describes a spe-

cific Pini-related service such as a reference of the distributed Pini-LUS. If this fails, it is determined, whether specific service references are specified - in this case, a preinstalled service proxy stub class is searched meeting the given service interfaces. In the most general case - if a `ServiceDescription` does not describe a Pini related service, and the Pini client is not limited to dedicated services - the Pini environment checks whether it finds a class that matches the interfaces of this service. It simply tests all combinations of interface names given in the `ServiceDescription` until either a service reference object can be initialized using the generated name or all combinations of interface names failed. This finally means that classes of such service reference objects must be available at the device in advance, in order to avoid dynamic remote class loading - they are synthesized from the service interfaces in advance by a special Pini compiler. Such service reference objects are called **service proxy stubs**, and they are fundamentally related to `PiniRMI`: A proxy stub class implements all service interfaces passed to the compiler. This means, the proxy stub is exactly of these types, which a demanded service should have.

Remark: `PiniRMI` stubs generally implement only a single interface that is related to the respective object - in contrast, service proxy stubs usually implement more than one interface.

Service proxy stub classes must be generated before a client is started - this does not contradict the plug-and-participate paradigm, since interfaces of services of interest must be known in advance anyway; otherwise, a service cannot be used. This provision of proxy stub classes, however, only requires one additional compile step, but no manual implementation of classes. That is, after specifying the service interfaces, they must be passed to a compiler in order to generate the classes. The compiler introspects the interface methods and generates appropriate code that implements these methods in the same fashion as when an arbitrary `PiniRMI` stub is synthesized. The name of this class is finally a combination of all interface names, which justifies the sketched procedure of instantiating a service proxy stub.

A further aspect of such proxy stubs compared with arbitrary stubs is that they implement several interfaces as already described. However, a proxy stub does not necessarily need to implement all interfaces of a particular service - if only a subset is implemented, the client can only access a subset of the service functionality. This leads to the complete independency of services and service users. For example, a dedicated service proxy stub class can be used for different services, if the interfaces match. In this way, the memory consumption of Pini applications can be be further reduced.

The Pini service use pattern is therefore rather simple: A service proxy stub represents the service on the client side, while the service itself runs a dedicated skeleton object. The client handles the proxy stub as the demanded service and performs all actions by invoking the corresponding methods. If a Pini service proxy stub receives a method call request for a service, it simply forwards the request via `PiniRMI` facilities to the corresponding service.

Such a service use pattern finally provides a limited device suitable plug-and-participate environment, where required resources, especially the required memory can be estimated in advance. This essential aspect will be evaluated in section 6.3. The Pini service proxy stub concept makes Pini obviously suitable for appling automation solutions on automation devices directly as described in [22], where the thesis' author presented Pini as a plug-and-participate technology for automation systems. The service proxy stub concept is one major contribution in this respect.

6.2.2.4 Robustness Concept of Pini

Pini's robustness concept generally relies on the fact that each registration with the community management system is annotated with a respective validity. If the registration is to be maintained for a longer time than specified by the validity, it must be renewed periodically. This incorporates both parties: the distributed Pini-LUS and the entities issuing a registration to the system. A registration is generally acknowledged by the LUS with a dedicated object, which contains a lease instance representing the validity duration. The Pini entity has to renew the lease periodically in order to keep the registration valid - this simply requires the `renew(..)` method of a lease object to be invoked.

These rather client related operations require compliant LUS counterparts: A LUS has to store the registration information together with their annotated lease objects, so that it can maintain these registrations with respect to their validity. If an entity renews the lease, the affected LUS must incorporate this into its list of registrations. Otherwise, if a lease duration expires, the LUS must remove the entry from its repository - this particular registration is no longer available in the community.

Aside from this rather general concept, a specific characteristic of Pini leases must be considered: they are `PiniRemoteObject` instances. The lease objects therefore are situated at LUSs, while entities holding a corresponding registration simply handle lease stubs. In this way, of course, resources are saved:

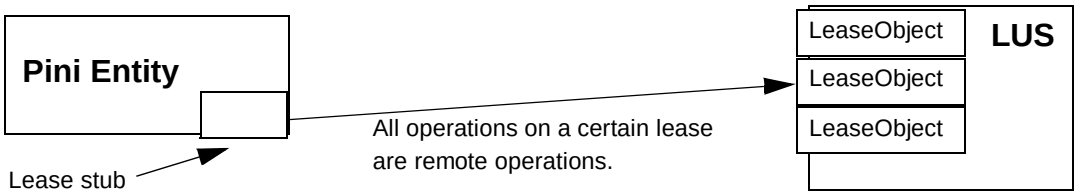


FIGURE 23. Basic architecture of the Pini lease concept

Jini, in contrast, handles leases as local objects and thus transfers entire objects to its community entities. This wastes resources, if the same object exists in several copies in the community - although both lease concepts rely on the same interface, Pini shows a more efficient and therefore more limited device suitable architecture than Jini.

6.2.2.5 Passive Information System

In addition to the service search concept of plug-and-participate technologies, representing an active mechanism to retrieve service references, also a passive information system is provided by plug-and-participate systems as defined in the abstract plug-and-participate interface and therefore also in Pini. In particular, the similarity to Jini leads to a remote event notification mechanism defined in Pini. Such a mechanism relies on the registration of interested entities within the community management system, which always has an up-to-date view of the infrastructure. This allows the distributed Pini-LUS architecture to inform interested parties if any change in the infrastructure occurs. Entities, therefore, must issue several data to the distributed Pini-LUSs in order to identify events of interest:

- a `ServiceTemplate` describes the services of interest

- the event type defines whether services join or leave the infrastructure or simply alter their attributes
- information about a reference to an event listener instance that finally receives occuring events

The information is sent to the LUS by invoking the `notify(..)` method of appropriate LUS reference objects. Requests are stored at the LUS, and acknowledgement messages are sent back to the requesting client. Such acknowledgement messages mainly contain the lease annotated with this registration.

This sketched procedure only covers one aspect of the passive information system, namely the registration procedure - the second aspect focuses on the generation and delivery of events: Pini's community management system always has an up-to-date view of the entire community, and thus can inform interested parties efficiently, if any change occurs. For example, if a service joins or leaves the infrastructure, the distributed Pini-LUS architecture is involved into this process and can inform registered parties about the modified infrastructure. The same applies for modified attributes - LUSs are always aware of changes in their repositories. In such cases, they create remote event objects and compare the event data such as `ServiceID`, `ServiceDescription` and attributes with the stored registrations for events. If a registration is found, which matches the event data, the LUS sends an event message to the given reference of the remote event listener of the corresponding entity. This procedure shows an essential difference to the Jini approach: While in Jini, a complete reference to the remote event listener is uploaded to the LUS - e.g, the entire listener object or a stub of it - in Pini, only a description is provided that contains all necessary information about the listener instance. Such an information is encapsulated in standardized and generic `RemoteEventDescription` instances. Using this information, the Pini-LUS establishes a connection to the listener based upon `PiniRMI`. In particular, each Pini remote event listener instance is of the type `PiniRemoteEventListener`, which in turn extends the `PiniRemoteObject` class (compare this also with the `DefaultPiniService` concept). Each remote event listener therefore transparently installs a skeleton and initializes the required `PiniRMI` infrastructure with the `MethodCallListener`. The `PiniRemoteEventListener` instance provides all required information for establishing a conncection to the listener for the event delivery - it is published within the `RemoteEventRegistration` object during the registration process. At the LUS, a dedicated stub class is situated, which is initialized when an event is to be delivered.

This concept relying on a description and a stub class, especially decouples interested Pini entities completely from the commmunity management layer; i.e., the dedicated stub class is generic, so that it can be used for each remote event registration and each event delivery process. Moreover, a LUS can install one permanent stub, which can be configured for different skeleton objects instead of running several stubs - this further saves the limited resources at the LUS. The most essential feature of this concept, however, concerns the availability of the stub classes - the `PiniRMI` stub class of the `PiniRemoteEventListener` is preinstalled at the Pini-LUS. Especially, only one class is required as justified in this paragraph.

This concept is adopted from the service provision, and is adapted to event delivery needs. For example, the `RemoteEventDescription` comprises nearly the same information as the `ServiceDescription`, such as the port number of the `MethodCallListener` instance and the host address. The interface information can be neglected, since it is standardized. Likewise, the basic architecture is similar: The `PiniRemoteEventListener` transparently initializes the underlying infrastructure, while

at the LUS, a stub is installed for event delivery (either on demand if an event is to be delivered, or one single permanent stub).

Although the concepts surveyed in this section rely on similar interfaces to Jini's counterpart, a more innovative structure results: Pini's passive information system completely decouples the interested parties from the community management simply by providing descriptions instead of complete reference objects. The descriptions are uploaded to the distributed Pini-LUS architecture but do not require any remote class loading as with Jini. Likewise, the inverse direction is also decoupled - the Pini-LUS initializes a pre-installed class in order to deliver the event - there is always the same class used for this purpose. This makes for a resource-saving facility that either installs communication objects on demand, namely stubs, or installs one single permanent stub that can be configured for each corresponding skeleton. It is in contrast to Jini's approach, in which a list of complete reference objects is maintained that requires dynamic remote class loading in order to restore the references - such a behavior is completely unpredictable in terms of memory required. The Jini concept is therefore not suitable for limited devices, especially in the field of automation.

6.2.2.6 Attributes

In addition to the provision of a unique identification of entities/services, as well as suitable access paths to entities, well-defined services must also provide a detailed description of their functionalities. The abstract plug-and-participate interface defines an attribute mechanism that provides a generic description of entities. In Pini, this attribute concept must function within the constraints given by limited devices, and must be as powerful and generic as possible. Pini's attribute mechanism therefore defines so-called `XMLEntry` objects that carry XML-formatted attributes of the type `XMLObject` (see also section 5.3.3.3.7 for such objects). This design decision relies on several motivational points:

- `XMLObject` structures represent tag-based trees which allow an efficient and powerful matching algorithm as described in section 5.3.3.3.7. For example, it allows range matching, exact matching as well as partial matching. In contrast, Jini's counterpart only supports exact matching, which is obviously insufficient for most fields of application - the Jini matching algorithm is surveyed in section 5.3.3.3.7, as well.
- `XMLObject` instances close the gap between today's automation-related description facilities and state-of-the-art plug-and-participate systems: While automation system vendors more and more often provide XML-formatted descriptions, current plug-and-participate systems do not necessarily support the matching of such descriptions.
- `XMLObject` instances representing tag-based trees are able to describe almost all types of data in an efficient manner. Such objects alleviate serialization and de-serialization processes due to their simple structure. They are well-suited for use in distributed systems, especially for systems relying on intensive data exchange and remote method invocation based on the object-oriented programming paradigm. Moreover, the `XMLObject` concept also makes the Pini attribute mechanism suitable for limited devices, since the tag-based tree approach provides a rather resource-efficient mechanism for data representation relying only on Java standard data types and data structures.

This basic representation mechanism for Pini service attributes is extended by appropriate mechanisms for attribute handling. Pini allows services to alter their attribute sets during runtime, but without removing the service from the community. The Pini attribute handling mechanism makes use of the similarity between Jini and Pini - services can modify their attributes via the `ServiceRegistration`

objects. These objects are returned by the LUSs as registration acknowledgements. For example, services can add new attributes, remove existing attributes and also modify them. `ServiceRegistration` instances provide dedicated methods for that purpose based on `PiniRMI` - remember that `ServiceRegistration` objects are simply stubs on the service side, while the corresponding remote object resides on the corresponding LUS entity.

Although this attribute mechanism of Pini is rather simple, it can be interpreted as a further important contribution to bring plug-and-participate to limited devices, especially automation devices: Avoiding remote class loading, but allowing efficient and precise descriptions as well as powerfull matching facilities are a crucial point in plug-and-participate - the service search facility fundamentally relies on the description of services and the comparison of requests with registrations. Moreover, using XML-based structures is an important step for closing the gap between state-of-the-art description in automation and matching facilities of classical plug-and-participate technologies.

6.3 Evaluation of Pini Facilities

The Pini concepts and protocols shown in the previous section provide an efficient plug-and-participate technology - this efficiency must be further evaluated, especially with respect to the intended target area of Pini, namely limited devices in the field of plant automation. This evaluation will be split into two parts: The evaluation of a simple Pini application compared with the Jini implementation of the same application in order to get a first impression of the resource consumption. Prior to this rather practical comparison, a conceptual evaluation of Pini will be provided. It comprises a classification of given concepts against further approaches and justifies particular design decisions, such as the decision to use broadcast communication in Pini's discovery protocol suite instead of multicast communication. This decision results from Pini's specific relation to automation systems, where broadcast as the basic group communication concept is more suitable:

- Broadcast communication is almost always provided by automation communication systems, while multicast is not necessarily defined.
- Industrial networks are rather isolated from each other such as to avoid foreign and unauthorized access due to inherent security and safety risks and also due to sensitive data in the system - the limited range of broadcast does not contradict these industrial requirements. In other words, messages do not cross network boundaries. These limitations make broadcast communication rather sufficient for such applications.
- The network load due to broadcast is justifiable, since only dedicated entities need to send broadcast messages periodically. In contrast, in UPnP-like networks, where almost all entities must send periodic (multicast) messages, the network load is considerably higher than that caused by Pini's broadcast-based discovery protocol suite and its distributed Pini-LUS architecture.

Moreover, moving the focus away from the industrial field and towards general applications on limited (and often mobile) devices such as handhelds, phones, etc., the limited range is implicitly compensated if the devices "themselves" cross network boundaries: Due to the distributed Pini-LUS architecture, services are always connected to the community, while clients can always contact all LUS entities

in their current network segment - the use of broadcast does not limit Pini's functionality nor stresses the network load disproportionally.

This robust and reliable behavior of Pini entities sketched in the last paragraph further explains the structure of Pini's community management facility as a distributed LUS. This architecture ensures that each service is at least registered with its self-provided LUS instance, while this LUS instance, and thus the registration, is transparent for the service.

Remark: If a service only performs the unicast discovery protocol or is limited to specific LUSs, it is not automatically/transparently registered with its local LUS module. The registration with the local LUS module is only in the general case if the broadcast discovery protocol is used.

In this way, each service can offer its functionality even if no central LUS is available. It is furthermore ensured that the community structure remains, even if a central LUS instance fails - **no single-point-of-failure facility** exists, in contrast to Jini. This can be further validated by the client behavior, because they can request the distributed LUS components for services of interest even if central LUS instances are not available. Clients can always retrieve references to services, while services are always able to announce their availability regardless of any central LUS.

Also the increased network load is justifiable: Although each service, more precisely the LUS component of each service, sends periodic broadcast announcement messages to the network, the load is still less than that caused by UPnP. Each service sends only a single message every 120 seconds - in contrast, each UPnP device sends at least four messages per period, although the periods are generally longer than in Pini.

Remark: These four UPnP advertisement messages result from the minimal UPnP configuration: one root device (3 messages) comprising one service (1 message).

However, such long UPnP periods increase the latency of the system - if an entity fails, it takes relatively long until this is noticed by the system. In Pini infrastructures, a nonfunctioning LUS will be detected after at most two minutes, while a crashed service will be detected after at most five minutes when its lease is expired.

Remark: This lease expiration depends on the lease duration usually granted by the LUS. It is generally limited to five minutes.

The service-related LUS components only provide limited functionality: They allow only local services to register in order to limit the management load, while the service search facility provides complete functionality. In contrast, the central LUS instances participate in every protocol of general plug-and-participate systems.

All these facilities of the hybrid community management system together define an innovative and efficient system. Its architecture results from the author's evaluation of UPnP requirements and specific experiments with Jini - the distributed Pini-LUS obviously combines advantages of both technologies while avoiding, or at least limiting, their shortcomings.

A further innovation of Pini concerns the provision of services and their service access paths via `ServiceDescription` instances and service proxy stubs, respectively. Both concepts together make Pini suitable for limited devices. For example, `ServiceDescription` instances contain the information about the service and specify how to contact and use the service. The `ServiceDescription`

concept is generic and is used throughout the entire system, regardless of the service to be provided. Moreover, the concept is also used in the passive information system in an adapted fashion - the `RemoteEventDescription` basically comprises the same information, which is encapsulated in descriptions and uploaded to the LUS. The compact format ensures a resource-efficient facility that always consumes a similar amount of memory.

Remark: The base size of such objects is relatively equal and represents the sum of its basic values:

- the hashcode of the skeleton object represented as an `int`-value
- the address of the service device as a `String`
- the port of the `MethodCallListener` as an `int`-value

In addition to these values, a `String` array consisting of the names of the interfaces is contained, which causes the size of `ServiceDescription` objects to be slighly higher than the aforementioned base size.

The information contained in a `ServiceDescription` object is used to instantiate the service proxy stub, which represents the service to the particular client. Its service proxy stub class must be provided in advance - only one additional compile step is required in order to synthesize the proxy stub class from the service interfaces. Of course, this additional step means additional effort in development - but it ensures a powerful plug-and-participate technology for limited devices. This concept, resulting from the `ServiceDescription` facility and the service proxy stub concept, finally leads to the major fundament of plug-and-participate for limited devices, especially for platforms in the field of industrial automation: The amount of memory required can be pre-estimated by evaluating the size of proxy stub classes and their referred classes. A recursive estimation algorithm for the size can defined; i.e., a recursion similar to that of the serialization is implemented - the calculation of the size of an object requires the (recursive) calculation of the size of included data fields. This size is accumulated with the size on disk the particular classes require - the classes are usually loaded into the virtual machine, which obviously requires memory.

Remark: The estimated values, however, will differ from the real values, since a virtual machine internally instantiates certain data structures in order to host an object and to maintain its executed objects. Exact values can be determined by initializing the service proxy stub classes.

Finally, the overall memory consumption depends on the number of used services - the calculated object sizes of proxy stub objects are simply accumulated. This leads to the following two formulae:

(I)

$$size(object) = \begin{cases} primitiveSize(object) & \text{if object is of a primitive data type} \\ \sum_{i=0}^{fields?object?} (size(object.field_i) + sizeOnDisk(object.field_i)) & \text{otherwise} \end{cases}$$

(II)

$$consumption = \sum_{i=0}^{\#services} size(service_i)$$

This possibility of calculating the required memory is the major innovation of Pini in the plug-and-participate field. It results from the `ServiceDescription` and service proxy stub concepts as secondary innovations. `ServiceDescription` objects can always be downloaded from the LUSs without dynamic remote class loading - this process is predictable. Furthermore, the service proxy stub classes are preinstalled (generated based on the interfaces) - their required memory can also be determined in advance.

The reasonability of the `ServiceDescription` facility is furthermore proven by a research group at the ETH Zurich: They built a technology called Mini that is rather similar to Pini. Mini is described in [43] and [44], and shows essential similarities to the Pini concepts as explained in [27]. Especially the `ServiceDescription` facility is identified by the team as the crux in providing plug-and-participate to limited devices. However, their concept mainly concentrates on the provision of services rather than on the use of them and therefore they do not define a reasonable mechanism for using services on limited devices as clients.

Aside from the Mini technology, which is rather directly related to Pini, there also exist proposals that allow limited devices to participate in Jini applications, such as the Jini Surrogate Architecture sketched in section 2.3.2.5, as well as the JMatos approach of PsiNaptic (see [89]). However, all of these solutions - Mini, the Jini Surrogate Architecture and JMatos - are rather service provision oriented and consequently do not provide completely stand-alone plug-and-participate frameworks, as Pini does.

In order to finalize this conceptual evaluation, Pini's main innovations and thus the major contribution of the author to "plug-and-participate on limited devices" will be summarized in the following list:

- Pini comprises an **abstraction layer of the network facilities**. This layer provides an extension of the KVM/CLDC library by a `java.nio`-like concept. Furthermore, the abstraction is the basis for

Pini's compliance with almost all network technologies, especially automation relevant network systems.

- The Pini framework defines an **efficient serialization mechanism** that allows the transmission of objects through the network in a transparent fashion, e.g., via the `PiniRMI` concept as method invocation parameters or return values.
- The **distributed Pini-LUS** is the basis for an efficient and reliable service infrastructure - it avoids disproportional network load and single point-of-failure facilities.
- The **`ServiceDescription`** concept is a further innovation providing all necessary information regarding the service access path. In particular, the concept avoids remote class loading, and thus is the basis for the exact determination of the resource consumption of the corresponding service.
- Pini's **efficient service use pattern via service proxy stubs** is a third innovation in plug-and-participate: It provides the service functionality transparently to the clients, but neither rely on remote class loading nor on RPC such as UPnP. The major innovation in this respect, however, is the possibility to **exactly determine the resource consumption of services in advance** - as the crucial point in the field of limited devices and in automation especially.
- The **Pini attribute concept** closes the gap between more and more XML-based automation system descriptions and the attribute mechanisms of classical plug-and-participate technologies. It furthermore allows for a rather efficient and powerful matching algorithm.

This evaluation of the innovative Pini concepts and the detailed classification of them is followed by a more practical examination given in the subsequent section. An initial experiment with Pini will be shown describing the functionality of the entire Pini concept, as well as the tremendously reduced resource consumption.

6.4 Pini Requirements and Suitable Platforms

The second part of the Pini evaluation is rather of a practical nature - it aims to determine Pini's requirements and suitable platforms. Examing these requirements involves the basic classes and concepts, and accompanied the determination of utilized Java classes. From these determined classes, the target platforms of Pini can be derived, especially it can be examined whether the intended target platform KVM/CLDC is met. The second aspect in this requirement analysis is the evaluation of the memory consumption by Pini itself as the crucial criterion in the field of limited devices: First, the disk space requirements of the Pini framework are measured and compared with the corrresponding Jini disk space consumption. It will demonstrate Pini's compact reference implementation, and shows one facet why Pini is more suitable for limited devices. As the second issue, this first impression will be underlined by the examination of a simple Pini application and the measurement of its resource consumption in terms of runtime memory and disk space. Both measured values are finally compared with measurements taken from a corresponding Jini implementation of the same application.

6.4.1 Examination of the Basic Resource Consumption

Pini was proven as a suitable limited-device platform in the previous section - at least from the conceptual point of view. This covered the innovations of Pini as well as the basic concepts. Such an evaluation must also take place in practice, as well as in the measurements of required resources such as disk space and runtime memory: The first value, namely the disk space consumption, will be determined by estimating the size of each of the Pini packages and accumulating the values. The result is then compared with the corresponding values for Jini. This evaluation furthermore provides an overview of Pini's package structure, as well as an explanation of the contents of the packages. Moreover, also the `proprietary` package consisting of the final implementation of the network abstraction will be considered, although it does not belong to the Pini core features. It is the basis for almost all features, and must anyhow be installed in order to run Pini. In contrast, classes such as the `PiniRMI` stub/skeleton compiler and the proxy stub compiler are neglected here, because they are not directly related to the Pini infrastructure:

Package	Description	Size on Disk (kB)
`communicator`	Provides necessary features of the RPC protocol, and thus is the basis for the `PiniRMI` package.	19.6
`discovery`	Provides all required features regarding the discovery facility.	45.3
`entry`	Consists of classes necessary for attribute handling.	7.2
`event`	Offers all basic functionalities for remote event handling.	14.6
`lease`	Contains all classes necessary for the Pini lease mechanism.	18.4
`lookup`	Offers all classes required for establishing service - service user infrastructures.	27.3
`LookupService`	This package encapsulates the LUS functionality. It consists of two parts - one for designated, fully-equipped LUSs, and a second for dedicated, decentralized LUS instances. These two parts are implicitly defined because the required features are contained in the same classes, which can be referred in two different modes.	37.5
`network`	Within this package the network abstraction interfaces are defined.	3.33
`PiniRMI`	Offers the `PiniRMI` facilities such as `PiniRemoteObject` and `DefaultPiniStub`.	2.02
`proprietary`	This package contains the specific implementation of interfaces definied in the `network` package, and thus the size of this package depends on the platform used. The measurement given here is based on a KVM/CLDC implementation.	9.33
`util`	Within this package all classes are stored that support the Pini environment with specific data structures and other helpful features.	26.6
	Sum:	**211.18**

TABLE 8. The disk space required by the Pini reference implementation

The disk space consumption of Pini will now be compared to its Jini counterpart - Jini's required disk space has already been evaluated in section 2.3.4, especially in table 1. It will be repeated here in a more detailed fashion by listing the Jini packages and archives accompanied with a brief description of their functions:

Package	Description	Size on Disk (kB)
Jini core	This package provides interfaces that define the basis of the Jini architecture. The package is aggregated in the `jini-core.jar` file of the Jini 1.2 distribution. Moreover, this package defines the basic interface suite on which both technologies rely.	`27 / 31.7`
Jini extension	This package provides the implementation of the interfaces defined in the Jini core package. It is stored in the `jini-ext.jar` file of the Jini 1.2 distribution.	`156 / 266`
LUS	Within this package all necessary classes of the LUS are aggregated. This basically applies to the non-standard package `com.sun.jini.reggie`, but are delivered in the `reggie.jar` archive.	`320 / 573`
LUS proxy	The classes of the Jini LUS proxy are also defined in the `com.sun.jini.reggie` package, while they are finally delivered in the `reggie-dl.jar` file.	`81.9 / 150`
RMI	The package contains all Java RMI related features of the SDK 1.4.0 runtime environment. Further packages and facilities RMI require, such as `java.lang.reflect` and `java.net`, are not further considered - they belong to the Java standard classes.	`-- / 83.4`
	Sum:	**`584.9 / 1104.1`**

TABLE 9. The disk space required by the Jini reference implementation

Remark: The values listed in the column "Size on Disk" represent the size of the jar-archive and the extracted classes, respectively.

Comparing the values measured for Pini's disk space consumption with Jini's values estimated for the extracted classes, one can obviously see that Pini consumes about one fifth that of Jini - this shows that Pini is more suitable for limited devices than Jini. However, when considering only the disk space consumption of both platforms and comparing them with the storage capacities of the devices listed in section 3.3, both Jini and Pini are suitable for limited device. As it will be seen, though, this applies solely to the disk space requirements - the examination of runtime and process memory consumption will show that only Pini is indeed applicable to limited devices and their platforms, while Jini requires considerable more RAM than limited or legacy devices can offer.

The examination of the runtime memory consumption of Pini requires the definition of a reasonable and representative application, which covers the major plug-and-participate concepts. This application must allow reliable measurements of the resource consumption of the technology infrastructures. Such a measurement must not be increased due to any complex service functionality. Therefore, a simple plug-and-participate application is defined, which offers a service, which accumulates two numbers. Likewise, a corresponding client is defined. The third component covered by this measurement is the

community management system of both systems, namely the distributed Pini-LUS and the central Jini-LUS. The measured values for both applications are listed in the following table:

Entity	Technology	Size on Disk (kB)	Runtime Memory (kB)	Process Memory (kB)
Service and proxy	Jini	7.45	379.05	5202
Service and skeleton	Pini	5.49	122.42	824
Client	Jini	3.22	522.59	5100
Client and proxy stub	Pini	3.72	111.61	936
LUS	Jini	--	906	21348
LUS	Pini	--	119.84	986
	Sum (Jini):	**10.67**	**1807.97**	**31650**
	Sum (Pini):	**9.21**	**353.87**	**2746**

TABLE 10. A comparison of resource consumption of Jini and Pini services

Comparing the values given for the column “size on disk” indicate the similarity of the applications - they are simply migrated from one technology to the other, while taking into consideration the special features of each technology. Although both applications require a rather similar amount of disk space and provide identical functions, their runtime consumption as well as their process memory differ greatly. These differences expressed in absolute values will be further investigated - generally they rest upon the differences in the mechanisms and protocols used.

Remark: The measurement of runtime memory is carried out using a specific tool, which is described in section 7.1 - it periodically measures the free memory and writes the value to a log file. The process memory, in contrast, was measured using the Task Manager of the Windows OS. Hence, the runtime memory comprises the consumption within a dedicated JVM, while the process memory incorporates all started processes required for properly running an application.

However, absolute values refer to specific applications, and must be abstracted in order to gain a common comparison basis. Moreover, abstracting these absolute values allows for the calculation of appropriate ratios between the technologies, which, lastly, allow a more objective evaluation. For example, the following table lists the ratios between the accumulated values regarding disk space consumption, runtime memory and process memory:

Category	Absolute Value	Abstracted Values
Disk space	9.21 : 10.67	~ 1 : 1
Runtime memory	353.87 : 1807.97	~ 1 : 5
Process memory	2746 : 31650	~ 1 : 11

TABLE 11. Comparison of resource consumption between Jini and Pini in general

These values give a first impression of the entire resource consumption and their respective ratios between Jini and Pini - it shows rather clearly that Jini is not well-suited for limited devices as Pini, which, in contrast, consumes obviously only a fraction of the memory required by the corresponding Jini application. However, entire infrastructures are seldom installed on one particular host - plug-and-

participate applications are usually distributed across several hosts, and therefore the single components must also be evaluated regarding their memory consumption. This, however, validates the ratios given in table 11. For example, in comparing the runtime memory consumption of clients and services with the values shown in table 11, a similar ratio can be observed - Jini consumes about five times more memory than Pini clients and services:

Category	Runtime	Process
service	1 : 3	1 : 5
client	1 : 5	1 : 5

TABLE 12. Comparison of service and client related resource consumption

The process memory ratios of clients and services, however, differ from that given for the entire infrastructure - for services and clients only a ratio of 1:5 is observed, while table 11 depicts a ratio of 1:11. This difference is reasonable because of the included LUS measurements in the ratio from table 11:

Category	Ratio
runtime	1 : 8
process	1 : 20

TABLE 13. Comparison of the LUS related resource consumption

Looking to table 13, which illustrates the memory consumption ratio of the community management architectures, shows that also the runtime consumption of Jini is significantly higher than that of Pini. More significant is the process memory consumption ratio - a Jini LUS consumes about 20 times more memory than Pini's distributed LUS architecture (the measured values concern a central LUS). Considering this ratio, the differences between the application ratios and the particular process memory ratios are reasonable.

Remark: The disproportionally high process memory requirements of the Jini LUS results from necessary components that must run in addition to the plain LUS functionality such as an HTTP server allowing the download of LUS proxy classes as well as the RMID process. Such additional processes must also be considered in the measurements.

Finally, the given ratios indicate one argument that, despite similar disk space requirements, Jini is not well-suited for limited devices such as automation devices - if the complexity of automation applications, e.g., PABADIS applications, is considered, the expected memory consumption of Jini will exceed the capacity of automation devices. Applications based on Pini, in contrast, fit within the capacity of such devices. However, this evaluation only covered a rather simple application; a more complex application is evaluated in section 7.2. This resource consumption evaluation, and the first proof of its suitability for automation devices, will be further validated by an investigation of Pini platforms and by a final comparison with Jini's platform requirements.

6.4.2 Examination of Pini Platforms

The evaluation of suitable Pini platforms requires the determination of basic facilities used by Pini components. For example, the components must be investigated with respect to the Java classes and features implemented: The major fundament of Pini is network communication through the Pini network

abstraction. The abstraction relies essentially on features provided by the `java.io` package, while its implementation relies on `javax.microedition.io` classes - they must not be considered as basic Pini classes! In particular, basic functionalities are required that are offered by the following classes:

- `InputStream`, `OutputStream`
- `DataInputStream`, `DataOutputStream`
- `ByteArrayInputStream`, `ByteArrayOutputStream`

Remark: The Pini network abstraction results in a similar framework as defined by the `java.nio` package. Both provide appropriate facilities for reading data from streams. Adopting the idea of `java.nio` to the KVM/CLDC will improve the platform independency of applications.

Following the Pini package structure as given in table 8, all required classes and functionalities can be determined - it will be seen that Pini only relies on a few standard Java classes: In addition to the `java.io` classes, Pini's network communication also requires facilities from the `java.lang` package, such as `String`, `Object`, `Exception`, `Class`, `Thread`, `System` and `Integer`. Moreover, Pini as a plug-and-participate technology mainly relies on (Java) network communication mechanisms - most of the Pini features require network access as already shown in table 8. A further investigation of all packages, which need not to be explained in detail, results in the list of required Java packages and classes - it is equal to the already given classes and packages. Only one exception exists; Pini uses features from the `java.util` package like `Hashtable` and `Vector` - both simply represent data structures and tools allowing an alleviated handling of data. However, they are not essential for Pini, because such facilities can be replaced by proprietary and customized structures in order to save further resources.

This rather overview-like but sufficient evaluation of Pini requirements will be more detailed by investigating a specific class and its required facilities: `Class`. Pini's service provision via `ServiceDescription` instances, its service use pattern via service proxy stubs as well as its serialization mechanism essentially rely on this class, especially the following methods are required:

- `public static Class forName(String name)`
- `public Object newInstance()`
- `public String getName()`
- `public boolean isInstance(Object o)`

Of course, the Pini concept relies on the functionality of these methods, while its reference implementation uses them directly. Another implementation may use other programming languages having other classes and therefore different methods.

This also applies to Pini's implementation of the network abstraction. The current reference implementation relies on the KVM/CLDC v1.1 and uses the Java extension package `javax.microedition.io` for the provision of networking mechanisms. For example, the following classes are used:

- `javax.microedition.io`
    - `Connector`
    - `Connection`
    - `Datagram`, `DatagramConnection`

A second implementation of this network abstraction is provided based on standard `java.net` classes. It basically uses the following classes:

- `java.net`
    - `Socket`
    - `ServerSocket`
    - `DatagramSocket`

The differences in these classes further explain the network abstraction of Pini - the plug-and-participate concept is provided completely independent of any underlying class library and networking facility.

Comparing the required classes and features (except the network abstraction implementation classes) with facilities, which are provided by different available Java platforms, the following list of Pini compliant platforms can be derived. They cover a wide range as shown below:

- J9
- Jamaica
- CVM/CDC
- PersonalJava
- Perc VM (J2SE 1.3/J2ME CDC
- Jeode (J2ME CDC)
- JWorks (JDK 1.1.8/1.2.2, as well as J2ME CDC)
- JTime (J2ME CVM/CDC 1.0)

Moreover, the listed platforms represent existing limited device and automation platforms; for example, the last four platforms are industry-relevant Java platforms (see table 3).

This evaluation finally classifies Pini as a limited device suitable plug-and-participate technology that is especially compliant with automation platforms. Jini, for example, cannot be applied to the listed platforms, or only with major re-design efforts, while Pini can be directly applied. Both technologies implement the abstract plug-and-participate interface, rely furthermore on similar interface suites, and thus can be classified as full-featured technologies - Pini is not a reduced Jini, despite its similar interfaces! Pini’s innovative architecture allows a wide range of fields of application - this will be evaluated in chapter 7 showing the combination of all three facets of this thesis, namely “*Plug-and-participate* for *limited devices* in the field of *industrial automation*.”

Moreover, further Pini aspects like a Pini-based agent platform are shown, as well as a gateway allowing the parallel application of Pini and Jini.

CHAPTER 7

Pini in Plant Automation

"Pini in Plant Automation" - this statement and chapter title cover all aspects of this thesis: Pini is a plug-and-participate technology for limited devices and is furthermore fundamentally related to automation systems. This chapter therefore combines all three facets of the thesis. For example, it describes a Pini-based CMU implementation illustrating the alleviated migration from Jini to Pini. It is furthermore shown that the Pini-based CMU is compliant to the automation platforms listed in section 3.3. A second example is a Pini-based Lightweight Agent Platform called PLAP being an automation related agent system fitting to state-of-the-art automation environments and following the trend to "agentize" automation solutions.

Both examples are the basis for measurements of the resource consumption, especially the memory consumption of the Pini-based CMU and a simple PLAP application. They will justify Pini's suitability for limited devices.

In additions to these rather automation-related topics, this section also covers a measurement tool, which is used for resource consumption determination. Furthermore, a gateway is presented that allows Jini and Pini to cooperate with each other - such a gateway can be the basis for integrated automation solutions, e.g., from the office level down to the field level in plants.

7.1 Measurement Tool

The measurement tool is not an essential part of the Pini solution, but it describes how to retrieve the measured values. It therefore underlines the quality of the measurements. An overview of this tool will show details of the data retrieval, and will furthermore alleviate the evaluation and the understanding of the measured values.

The tool is used in all applications such as LUSs, the CMUs or arbitrary Pini clients and services, in order to determine their runtime memory consumption. It is started before the respective application is launched in order to obtain an offset of available memory. Based upon this offset, the consumption can be calculated, if the current available memory is periodically determined - the difference between the offset and the current value makes up the consumption. The tool therefore runs in the same JVM as the application - parallel in a dedicated `Thread` - determines the current available memory and writes it to a log-file. This a mechanism is illustrated in code example 33:

```
public void run(){
      while(true){
      try{
        long freeMem = runtime.freeMemory();
        dos.writeLong(freeMem);
        }catch(Exception e){…}
      }
}
```

CODE EXAMPLE 33. Memory track thread - periodic measurement of free memory

This also explains the determination of the offset value before the application is started, but after the tool is launched. The tool itself also consumes memory, which must be excluded from the measurements.

A further aspect aside from the plain measurement functionality refers to the logging of values: Limited device platforms not necessarily define file access, but instead network access mechanisms. The tool therefore contacts a server, which manages the log files. This furthermore ensures that no additional memory is wasted due to buffered write requests, since network access is usually processed immediately without further buffering.

After finishing the measurements, the log-files are evaluated by calculating the average of all measured values. For example, the runtime memory consumption values given in section 6.4 were measured in this way. The tool is furthermore the basis of those measurements shown in the following sections covering a Pini-based PABADIS CMU implementation and a lightweight agent platform. However, due to Java's unpredictable garbage collection algorithm the measurement can vary from the exact value.

7.2 The Pini-Based PABADIS CMU

The major focus of this chapter is on the application of limited device compliant plug-and-participate technologies in automation - an example for this a Pini-based PABADIS CMU. It covers especially four aspects with respect to Pini and plant automation. First, the easiness of migrating Jini-based applications to Pini is shown as the final proof of the associated design decision regarding the used interface suites. Second, the measurement of resources and a further evidence of Pini's suitability for limited device is given. The third aspect is with respect to the PABADIS solution - it shows that PABADIS can be applied to automation systems directly, while the current Jini-based reference implementation only allows COM-linked as well as OS-linked CMUs. Pini will furthermore allow Java-based CMUs. This covers the fourth aspect of Pini as the basis for rather complex applications. The Pini-based CMU reference implementation describes an example for this direct application of automation solutions on automation devices, which results in a fundamental benefit of Pini: It ensures rather efficient and effective automation solutions, especially distributed automation solutions. This generally refers to the architecture of automation devices - they usually consist of many controller components, which usually have limited-device character. If solutions like PABADIS can be applied to automation devices directly, this will avoid additional hardware, which would be required if state-of-the-art technologies were used. Pini therefore can increase the added value of devices - it is a basic technology, which supports the paradigm change from centralized automation to distributed solutions as a fundamental evolutionary step.

Following the objective of this section, and coping with the structure of CMUs, the migration from a Jini-based CMU to a Pini-based CMU comprises two main steps: First, the migration of the `PnPModule` towards Pini, and second the adaptation of further components to the new platform KVM/CLDC. This migration is the basis for measurements, which finally underline Pini as an automation-related plug-and-participate technology.

7.2.1 A Pini-Based `PnPModule`

The implementation of a Pini-based CMU mainly concentrates on the re-design of the `PnPModule` package. It is implemented based on Jini in the current PABADIS reference implementation, and thus the migration from Jini to Pini is rather simple. Several classes must be updated - the main focus in this respect is on the `PnPModule` class comprising the remote event mechanism, the join and lookup facilities, lease mechanism and attribute handling.

While other classes of the package simply require an update of the import statements, the `PnPModule` class requires specific modifications. It implements the automation related plug-and-participate interface, and thus is the essential module of this package. However, the modifications are rather simple:

- The join mechanism must be re-designed with respect to the provision of a `DefaultPiniService` object instead of a service proxy.
- Adapting the remote event handling to the Pini-specific features.
- Definition of CMUs as Pini clients, namely as `DefaultPiniClient` instances.

- The adaptation of the lookup mechanism in terms of providing PABADIS related `Matches` objects based upon the retrieved `ServiceDescription` objects.
- General adaptation of classes to the KVM/CLDC platform.

Within the following subsections, these facilities will be described in more detail - however, most of the concepts are rather similar to the Jini-based implementation, and thus only essential modifications need to be sketched.

7.2.1.1 Pini CMU - The Join Mechanism

The objective of the join mechanism is always the same: Announcing the availability of CMUs and their functions in the PABADIS community. A CMU therefore must publish specific information such as its `ServiceID` (if already assigned), its attributes, and the service access path - in the case of Pini, this refers to the `ServiceDescription`. In Jini, in contrast, `FunctionProxy` instances are provided as service proxies. These objects represent the CMU in the community and allow the connection with the RA. This `FunctionProxy` class can be assumed as the basic plug-and-participate service of each CMU, and therefore they are adopted to represent a Pini service. The Pini-based `FunctionProxy` implementation consequently extends the `DefaultPiniService` class, while the implementation of the class remains in its general form:

```
public class FunctionProxy extends DefaultPiniService implements IFunc-
tionProxy{
  . . .
}
```

CODE EXAMPLE 34. The adapted `FunctionProxy` of the Pini CMU

This also requires the definition of a skeleton, which is automatically generated by the `PiniRMI` compiler by passing the `IFunctionProxy` interface. Such a simple adaptation of a CMU (represented as a Jini service) to a Pini service completes the migration of the join mechanism from Jini to Pini:

```
for (int i=0;i<item.length;i++){
 item[i]=new ServiceItem(null, functionproxy[i],
                         new XMLEntry[]{attr[i]});
}
. . .
for (int i=0;i<item.length;i++){
  try{
    registration = registrar.register(item[i],leaseDuration);
    . . .
  }catch(Exception e){}
}
```

CODE EXAMPLE 35. The adapted join mechanism of the Pini CMU

As defined for the Jini implementation, the `FunctionProxy` objects are passed to the `ServiceItem` objects to be created - in this Pini-related context they represent the Pini service, while in Jini they simply represent the proxy of the function. The registration is the same, since the adopted `register(..)` method. The `ServiceRegistration` objects returned from the LUS are finally stored in a list so that the attribute handling is ensured, as well as the lease object is obtained and passed to the (Pini-related) `PABADISLeaseRenewalManager` - this structure does not need any modification.

Obviously, the migration of the CMU join mechanism is rather simple, and affects only a few parts in the `PnPModule` class - the same applies for function search facilities, which is shown in the following section.

7.2.1.2 Pini CMU - Function Search

The adaptation of the CMU's function search facility simply refers to the initialization of the `FunctionProxy` reference objects. They are created by invoking the `getServiceReferenceObject()` method of the corresponding `ServiceDescription` object retrieved from the distributed Pini-LUS architecture by a lookup request. This means, a `ServiceTemplate` object is created based on the given `SearchPattern` object, which is finally passed to the distributed Pini-LUS - the rest of the function search facility remains in its form:

```
ServiceTemplate template = new ServiceTemplate(null,
                                        pattern.getTypes(),attr);

while (lusCount<registrars.length && temp.total<maxMatches){
  //perform the lookup protocol via calling the lookup method
  ServiceMatches jmatches =
                    registrars[lusCount].lookup(template,maxMatches);
  . . .
}
```

CODE EXAMPLE 36. The lookup facility of the Pini CMU

The preparation of the PABADIS `Matches` objects must be adapted in order to instantiate the corresponding service references defined by the retrieved `ServiceDescription` objects. In turn, the `ServiceDescription` objects are obtained from the `ServiceItem` instances carried by `ServiceMatches`, which are the result of the lookup request:

```
int count=0;
  while (count<jmatches.totalMatches && temp.total<maxMatches){
     Object service = jmatches.items[count].getServiceDescription().
                                                 getServiceReferenceObject();
     //check, if the found PnP service is a valid PABADIS entity.
     if (service instanceof IFunctionProxy){
          //Evaluate the contained information
                 . . .
     }
  }
```

CODE EXAMPLE 37. Evaluation of retrieved `ServiceMatches` to PABADIS `Matches`

After such service references are obtained, the corresponding `Matches` are created in the same fashion as already shown for the Jini-based reference implementation. These objects are returned via the CMU's RA to the requesting entity, and the general PABADIS logic is kept.

7.2.1.3 Pini CMU - Remote Event Facility

In the same way as the Jini-based PABADIS CMU, also the Pini counterpart defines a remote event notification mechanism. This facility can be directly migrated to the Pini technology, and likewise focuses on two distinct aspects:

- the registration and the plug-and-participate concerned part, and
- the dispatching procedure of events to their addressed final receivers, which is out of the scope of the plug-and-participate framework.

The first topic is a rather simple migration of the program statements under consideration of specific Pini features, while the final dispatching is interlocked with such Pini-specific features, especially with `PiniRemoteEventListener` instances:

Migrating the remote event facility to Pini mainly concerns the modified signature of the `notify(..)` method. Most essential in this respect is the provision of a description of the listener instances that are used for establishing a connection to described listeners in order to dispatch occuring events - instead of a reference object, a `RemoteEventDescription` is sent to the LUS. Moreover, the hand-back object is removed:

```
try{
   //perform the notification
   EventRegistration evReg=registrar.notify(template,
                                   transition,
                                   listener.getDescription(),
                                   leaseDuration);
}catch(Exception e){...}
```

CODE EXAMPLE 38. The registration for remote event notification is similar to its Jini counterpart

The required description object is retrieved from the `PiniRemoteEventListener` instance as shown in the previous code example. In this respect, it describes the major difference between the Pini event system and its Jini counterpart: Each remote event listener must extend the `PiniRemoteEventListener` class, so that the required underlying infrastructure is initialized transparently (see also section 6.2.2.5). Via this listener, the events are received when they are delivered from the LUSs - seen from the plug-and-participate technology, the event is dispatched to the final receiver. However, the particular CMU infrastructure defines dedicated PABADIS entities as final receivers such as PAs, PMAs, or even RAs. Coping with this problem leads to a specific PABADIS remote event architecture based on the Pini concept: Each registration request to the `PnPModule` results in the installation of a separate listener instance. Although many listener instances might possibly be installed, only one single `MethodCallListener` is installed for all these `PiniRemoteEventListener` instances. These instances are annotated with an ID, which is used in order to distinguish among different registrations of different PABADIS entities during the event dispatching:

```
class PRemoteEventListener extends PiniRemoteEventListener{

  public PRemoteEventListener(long id, EventNotifier notifier)
                                                 throws RemoteException{
    this.requestorID = id;
    this.notifier = notifier;
  }

  public void notify(RemoteEvent event)throws RemoteException{
    if (event instanceof ServiceEvent){
         //Evaluate the event respectively
         . . .
    }
  }
}
```

CODE EXAMPLE 39. The Pini CMU object to be notified about remote events

If an event occurs, the distributed Pini-LUS delivers it to the corresponding listener using the information from the `RemoteEventDescription`. The listener instance as the final plug-and-participate related receiver internally evaluates the event, and dispatches it to the PABADIS-related final receiver - annotated with the ID of the receiver, so that the RA is able to deliver the event appropriately.

This section 7.2.1 has shown an overview of the migration from the Jini-based PABADIS CMU to a Pini-based implementation - such a migration of a rather complex application finally validates the design decision regarding the interface suites. The result is a limited device compliant PABADIS CMU - at least seen from the plug-and-participate technology's point of view. However, due to the migration from fully-featured JVM platforms to the limited device platform KVM/CLDC, specific adaptations of further CMU components are required:

7.2.2 Pini CMU - Adaptation of Components

The migration of the CMU implementation from Jini to Pini is accompanied with a platform change - from fully-equipped JVMs to the limited-device platform KVM/CLDC. This requires specific migration efforts regarding the CMU components, although they have already been developed according to limited-device constraints. Neither the CMU concept nor its basic architecture is affected, but particular implementation issues. Some components must be adapted, which use facilities which are unavailable at the KVM/CLDC. These required modifications are described as a survey of the CMU structure:

- **Residential Agent:** The general CMU-related RA functionality is preserved, so that the RA basically acts as the general logic of the CMU - it performs its respective control tasks. However, all agent-related features must be removed because of the lack of a suitable limited-device agent platform.
- **Common Feature Module:** The CFM structure is preserved, und thus essential features such as MES tools, schedule and `PnPModule` are accessible in the defined fashion. Features such as memory manager, data management, tracer, security, HTTP, and SCADA have been neglected, since they obviously do not belong to the essential CMU features. Likewise, the XML package is removed because no limited-device-compliant XML parser is available, which furthermore satisfies the PABADIS requirements (see [71] for these requirements). This, however, impacts one essential PABADIS feature, namely the CD - the Pini CMU simply uses a predefined `XMLObject` representing a CD, which is perfectly sufficient for the demonstration purposes.
- **RAFI:** One of the most fundamental facets of PABADIS along with the provision of the distributed automation solution is the provision of the generic access path of automation functions. This facility, of course, is preserved in the KVM/CLDC-based CMU implementation. In particular, the essential components must be adapted in their implementation, while their logic has been kept:
    - Function Control Module
    - Event Control Module
    - Capability Description

  FCM and ECM required a re-design of their implementation in order to save memory: The PABADIS reference implementation controls the behavior of the finite state machines using an appropriate transition table, which consists of objects representing the particular transition. The KVM/CLDC implementation, however, substitutes this table with a compliant `switch`-statement - the required

functionality is preserved, and thus the generic access of functions is ensured. A further result of this re-design is a memory efficient solution with respect to disk space and runtime memory - neither an array must be allocated nor particular transition classes are required as described for PABADIS in [74].

Likewise the Capability Description facility has been adapted. This module provides a simulated CD, which is predefined for this demonstration purpose.

- **Agent Abstraction:** The Pini-based CMU implementation does not use the agent abstraction, as there is no suitable agent platform available that meets the PABADIS requirements.

Although the described adaptations reflect essential modifications of the entire CMU implementation, this variant is able to offer all basic CMU facilities like

- community join and representation of functions in the PABADIS infrastructure
- generic access of underlying functions via FCM and ECM
- proper participation in the PABADIS logic; i.e., the provision of an overview of the function infrastructure and remote event facility
- abstracted agent behavior - the Pini-based PABADIS CMU defines a RA class that is de-coupled from any agent feature. Its general behavior, however, is preserved.

Recalling the basic objective of this chapter - the integration of all three facets of this thesis - this Pini-based CMU is the final result in this concern - it allows the application of a revolutionary automation solution on automation devices directly. In particular, this simultaneously covers two innovations in the fields automation and computer science:

(1) Pini is a limited-device compliant plug-and-participate technology relying on innovative facilities such as the distributed Pini-LUS, `ServiceDescription`, and the service proxy stub concept for efficient service use. The essential aspect in this respect is the possibility to estimate the expected memory consumption of applications in advance as one major fundament for limited devices.

(2) Pini is one major fundament for migrating revolutionary automation applications like PABADIS to limited devices of the automation field. The innovation in this lies, on the one hand, in the direct application on the devices and, on the other hand, in the solution itself - it increases the efficiency and added value of such machines.

Both innovations are also covered in the following section, which finally evaluates the solution with respect to its resource consumption. It finally proves that both solutions are limited-device compliant:

7.2.3 Evaluation of Resource Consumption

The evaluation of the resource consumption of Pini-based CMUs basically refers to the measurement of memory consumed during runtime as well as its disk space consumption. These measurements will be compared with a respective Jini-based implementation - this requires a backwards adaptation from Pini to Jini. It ensures an objective comparison, since both variants rely on the same basic modifications. In particular, this backwards migration simply affects the `PnPModule` package, while other CMU components are not touched. The resulting CMU variants are rather equivalent and therefore consume a similar amount of disk space as illustrated in table 14 - it also shows the size of the PABADIS CMU

reference implementation with its fully-featured compontents such as HTTP server, XML parser, RA class, etc.:

Technology/CMU	Size on Disk (kB)
Jini CMU	176
Pini CMU	177
PABADIS reference implementation	751.17

TABLE 14. Disk space consumption of CMU variants

Although both CMU variants have nearly the same size on disk, and have similar implementations and offer identical functionalities relying on different basic technologies, their memory consumption regarding runtime memory and their process environments differ tremendously:

Technology	Runtime Memory (kB)	Process Memory (kB)
Jini CMU	~460	10500
Pini CMU	~230	850 - 1100
PABADIS reference implementation	~1218	22000

TABLE 15. Runtime and process memory consumption of Pini and Jini CMUs

Remark: The LUS facility of Pini and Jini is neglected, since the measurements simply consider single CMUs in the PABADIS infrastructure in order to focus on CMU consumption rather than the system consumption.

The given absolute values will be abstracted in order to allow for appropriate comparisons. Moreover, the absolute values are also used to examine whether the solutions would fit the automation device requirements shown in section 3.3, while the abstracted values are used to illustrate the rations between Jini and Pini.

Starting the evaluation with the absolute values shows that both CMU variants require less than 1 MB runtime memory - this obviously is compliant with such devices shown in table 2.

Remark: The evaluated devices provide RAM ranging from 4 MB up to 128 MB

Plain runtime memory measurements, however, only concern themselves with the consumption within specific runtime environments, e.g., in a single JVM. In particular, this measurement basically describes the consumption of objects within their respective VM: The Jini-based CMU implementation consumes about twice the memory of the Pini-based variant, although “exactly” the same implementation is used:

Measured Value	Absolute Value Pini : Jini	Ratio Pini : Jini
Runtime Memory	230 : 460	1 : 2

TABLE 16. Ratio of runtime memory consumptions between Pini and Jini CMUs

This abstraction and the resulting ratio of 1:2 is slightly contrary to table 12, where ratios between 1:3 and 1:5 are observed for services and clients. Such a difference basically relies on the distinction between dedicated clients and service applications, as given in section 6.4.1 - CMUs simultaneously

behave as clients and services. Although the given ratio of 1:2 of Pini- and Jini-based CMUs only describes a minor advantage of Pini in the migration of automation solutions to automation devices, the measured absolute values prove PABADIS as a limited-device-compliant solution. This means, the absolute memory consumption of PABADIS is rather low, and fulfills the prerequisites of automation devices.

However, while the runtime consumption does not sufficiently justify Pini as the fundament for applying revolutionary automation concepts to automation devices directly, the process memory consumption does:

Measured Value	Absolute Value Pini : Jini	Ratio Pini : Jini
Process memory	1100 : 10500	1 : 10

TABLE 17. Ratio of the process memory consumption between Pini and Jini CMUs

Comparing especially the absolute values with the prerequisites of state-of-the-art automation devices as given in table 2, it shows that the Pini-based CMU variant covers all devices. The Jini-based counterpart, however, cannot be applied to at least two of them. This obviously limits the field of application of this CMU variant, and thus may likewise impact the acceptance of such a solution. Moreover, if the current PABADIS reference implementation is considered, only three devices are finally in the target area - at least as for as memory consumption is concerned; regarding the provided platforms, the Jini-based CMU variant can be applied neither to any device nor to their provided platforms.

Remark: While Pini covers 100% of the automation devices shown in table 2, the corresponding Jini-based CMU variant only covers 78%. The PABADIS reference implementation, by comparison, only covers 33%. These rations show a significant advantage of Pini, since the devices of table 2 are representative state-of-the-art devices - they are a relevant random sampling of the entire field so that the given rations can be assumed as reasonable representative.

Moving the focus from the absolute values towards the abstracted values and the resulting ratios, Pini's specific contribution for applying distributed automation concepts like PABADIS directly to automation devices becomes clear: The Pini-based variant consumes only one 10-th that of the Jini-based variant. Such a significant ratio especially results from the fact that the Pini-based CMU runs all components within one dedicated process, while the Jini-based counterpart requires several additional facilities like `SecurityManager` instances and specific RMI processes. This allows Pini to provide a more efficient solution - the most fundamental argument, however, is the broad field of application: Pini ensures the application of automation solutions on almost all automation devices, which broadens the acceptance of revolutionary solutions. For example, in order to equip a plant with a Pini-based PABADIS system no specific devices need to be installed - contrary to a Jini-based solution. This is the key factor for bringing new solutions to markets, and Pini can be assumed as a basic fundament in this respect.

Finally, the measured values for memory consumption define a kind of offset value - the CMU is completely initialized and waits for PAs. In particular, the CMU offered all its functions to the PABADIS community, and thus behaves as a Pini service. PAs, in turn, will use the CMU functions for task processing, and in order to obtain an overview of the community. This means, a CMU will behave as a Pini client, and requests the distributed Pini-LUS for functions of interest. It downloads the accompanied `ServiceDescription` objects, which are used to instantiate corresponding service reference objects. These service reference objects are the service proxy stub objects of the corresponding

FunctionProxy instances situated at the corresponding CMUs. With respect to this Pini client related behavior, the memory consumption of the Pini services, namely of the FunctionProxy service reference objects, will be estimated: This calculation is defined by an accumulation of the given offset value and calculated values for retrieved service references using the formulae given in section 6.3. For example, a IFunctionProxy_PPStub object contains three fields, namely two int-values and a String carrying the address of the respective service. This results in an estimated consumption of 23 bytes for the fields. The size on disk is about 1.56 kB - finally an estimated consumption of 1.58 kB results. Measuring the consumption of a IFunctionProxy_PPStub object in the KVM, however, results in about 5.2 kB consumption - three times more than estimated. Such variation results from the dedicated internal structure within the virtual machine. However, the magnitude of the consumption can be estimated via the given formulae, while an exact determination is made possible by instantiating the available service proxy stub class in advance to measure its consumption - the claim to determine the consumed memory in advance is held.

7.3 A Pini-Based Lightweight Agent Platform

While the Pini-based CMU is directly related to automation systems, the following Pini-based Lightweight Agent Platform (PLAP) is only indirectly related: PLAP can be used as the basis for PABADIS infrastructures - this would save memory, since only one framework providing agent and plug-and-participate facilities simultaneously needs to be installed instead of two different technologies, namely a separate agent technology and a plug-and-participate technology. PLAP would allow the complete application of the PABADIS solution to automation devices - also the agent facilities can be migrated. This shows a second relation of PLAP and automation, since it follows the trend of “agentification” of automation solutions - PLAP allows the application of such solutions directly on automation devices.

Finally, PLAP is a complex Pini application, which provides an agent platform supporting mobility even on limited devices and their compliant platforms like the KVM/CLDC. This can be seen as an innovation in this area.

7.3.1 Motivation

In PABADIS the necessity exists for such an agent platform - first, in order to apply PABADIS finally to automation devices, and second in order to conserve resources such as disk space, runtime and process memory. However, PLAP is not the first effort in this field; i.e., there already exist platforms, which either are fully-equipped agent platforms, or which are based on plug-and-participate technologies, or which are limited device compliant, but suffer from a lack of mobility. The relevant platforms in this respect are LEAP [101], Ronin [15], or Dejay [100], which have been evaluated during the PABADIS research and the development of PLAP. The platforms are

- based on a plug-and-participate technology: Ronin is based on Jini and is therefore not suitable for limited devices.
- limited device suitable: LEAP is designed for limited devices, but lacks mobility of agents.
- an arbitrary agent platform: Dejay is a dialect of Java allowing to design agent applications - it is, however, not limited device compliant.

These platforms obviously fulfill neither PABADIS requirements nor meet the KVM/CLDC constraints, which both are the design goals of PLAP. Moreover, a second aspect of the specification lies in the consideration of general agent platform facilities, especially the concentration on such features and the consideration of automation needs. While general automation needs have been defined in [78] and aggregated in the PABADIS agent abstraction such as mobility, agent search, cooperation and communication, etc., general agent facilities are given by Jim Waldo in [104]:

1. use of active objects - objects travel around the network in order to fulfill their mission
2. interaction with various environments
3. ability to obtain information for its "own" reasons or on behalf of its owner
4. report the results of the work which was done
5. ability to make decisions on behalf of its owner

These listed facilities of agents are more general than automation needs - they especially do not contradict each other. PLAP therefore concentrates on these given features and provides an automation-compliant platform. General PLAP components are implied by these listed features:

- **Agents:** Objects, which behave on behalf of their owners and fulfill their particular mission autonomously using features of their environment.
- **Agent Host:** The environment for agents, in which they can perform tasks and where they have access to appropriate facilities required to complete the aforementioned tasks.
- **Lookup facility:** It provides the system with information about the infrastructure, such as existing agent hosts and active agents.

These given basic components are designed in essential relation to Pini: The lookup facility is provided by the Pini-LUS, while PLAP agents are basic Pini services. PLAP agent hosts are both Pini services and clients - a detailed view is given in the following section.

7.3.2 PLAP Details

Details on PLAP basically cover the PLAP agent hosts and PLAP agents - the LUS components are implicitly provided by the distributed Pini-LUS, which is accessed via the PLAP agent host component. Hence, this section has two particular focuses resulting in two subsections.

7.3.2.1 Architecture of PLAP Agent Hosts

An agent host is the environment in which agents usually reside. Such hosts must provide all necessary tools/facilities for agents required to perform their tasks. These facilities are aggregated within three main components of an agent host:

- Agent Management System (AMS)
- Agent Communication Channel (ACC)
- Standard Directory Facilitator (SDF)

Components like these describe FIPA compliant agent platforms (see also [35]). PLAP implements these components utilizing appropriate Pini concepts. For example, the SDF is given by a specific access path to the distributed Pini-LUS architecture. Likewise, the AMS component provides an

access path to the underlying community management system. These fundamental features are defined in two interfaces implemented by each PLAP agent host:

- `AgentHostGlobal`
- `AgentHostLocal`

The `AgentHostGlobal` interface offers facilities, which are required by agents and agent hosts in order to cooperate with each other:

```
public interface AgentHostGlobal{

  public String getHostAddress()throws RemoteException;

  public void communicate(ServiceID target, Message mes)
                                                throws RemoteException;

  public void migrateTo(ServiceID id,EventRegistration registr,
                          MobileAgent reference) throws RemoteException;
}
```

CODE EXAMPLE 40. The `AgentHostGlobal` interface

It describes the Pini service-related facilities of a PLAP agent host: Each PLAP agent host represents a Pini service providing appropriate facilities so that other PLAP entities can access them via the Pini service proxy stub concept. For example, the `getHostAddress()` method provides the host address in order to identify the host in the network, while the `communicate(..)` method allows the communication between agents as specified in the ACC. This communication path moreover is only one path out of three in FIPA-compliant platforms - a detailed description of the ACC and the communication paths is given in the agent related section 7.3.2.2.

These two interface methods are usually invoked by agents in order to participate in a PLAP community. Contrary to this, the `migrateTo(..)` method cannot be invoked by agents directly even though it supports the mobility of agents - it is usually invoked by agent hosts in order to initiate the migration of an agent. Agent migration requires specific preparation of the involved agent, and thus if an agent invokes this method directly without the required preparation, an exception will be thrown and the migration fails.

Communication and migration are the most fundamental concepts of the `AgentHostGlobal` interface. They essentially rely on Pini features such as the Pini services use concept, since the `AgentHostGlobal` interface specifies functionalities, which are are globally available as Pini service. If, for example, an agent retrieves a reference to a (remote) agent host via the lookup facility, it receives a proxy stub object, which implements this interface. An agent - or more abstract, a PLAP entity - therefore accesses this proxy stub. It finally forwards the requests to the corresponding agent

host instance. This means that this agent host instance behaves like a Pini service, and thus extends the `DefaultPiniService` class - the required `PiniRMI` infrastructure is initialized transparently:

```
public class AgentHostGlobal_impl extends DefaultPiniService
                                                implements AgentHostGlobal{
   public String getHostAddress()throws RemoteException{
     return getAddress();
   }

   public void communicate(ServiceID target, Message mes)
                                                    throws RemoteException{
      //lookup for the agent and deliver the message
      . . .
   }

   public void migrateTo(ServiceID id,EventNotifier registr,
                         MobileAgent reference) throws RemoteException{
      //check, whether the agent has been prepared for the turn
      . . .
      //start the migration Thread
      (new MigrationThread(…)).start();
   }
}
```

CODE EXAMPLE 41. The Pini service of each PLAP agent host

Such a behavior as a Pini service requires that the agent host itself joins the underlying Pini architecture in the defined manner: The host performs the Pini discovery protocol in order to obtain the LUS references. Subsequently, it joins the community and performs all necessary operations in order to maintain its registrations. Its `ServiceDescription` represents a Pini service, which implements the `AgentHostGlobal` interface. This general behavior as a Pini service allows the connection to the corresponding service facility.

If a PLAP entity invokes the `communicate(..)` method of an agent host reference object, the corresponding host receives this request via `PiniRMI` and then dispatches the message to the addressed agent. This process is rather simple - each agent host maintains a list of agents currently residing on it. From this list, the addressed agent reference is obtained based on the `ServiceID` transmitted as the agent ID.

The migration, in contrast, is more complex, since it defines a more complex concept and incoporates several components of the PLAP architecture. In particular, an agent usually defines its target host and obtains a reference to this host. The agent subsequently requests its current local agent host to start the migration process - the agent invokes the `migrateTo(..)` method of the `AgentHostLocal` interface described later in this section. This request requires a reference to the agent and the reference object of the target host - after a dedicated preparation (see the corresponding paragraphs later in this section), the local agent host invokes the `migrateTo(..)` method of the provided reference

object of the target host, which expects the agent reference. The request is forwarded to the corresponding (remote) agent host via `PiniRMI`, where the agent must be resumed: All its data structures will be restored, and the agent is re-started. These steps are implemented in a dedicated module of the `AgentHostGlobal` implementation. In particular, a `Thread` is launched that takes care of the processing of the migration request:

```
private class MigrationThread extends Thread{

      public void run(){
        try{
          //first set the agent id
          reference.setAgentID(id);
          //set the configuration
          reference.setConfig(agentConfig);
          //initialize the default Pini service
        ((DefaultPiniService)reference).init(agentConfig,reference);
          //then register the agent with the LUSs
          localHost.registerAfterMigration(reference);
          //register the agent for remote events...
          localHost.registerRemoteEvents(reference);
          //invoke afterMove
          ((MobileAgent)reference).afterMove();
          //start the agent
          ((Agent)reference).main();
        }catch(Exception e){ . . . }
      }
}
```

CODE EXAMPLE 42. `MigrationThread` implemented in the `AgentHostGlobal`-related component

This code example illustrates the detailed migration process on the target host: The agent ID must be set and the agent must be configured appropriately. Likewise, it must join the community in order to be available to other agents. Subsequently, the agent performs (user-specific) actions associated with a migration, and afterwards the agent is re-started by invoking its `main()` method. Furthermore, the reference to the local agent host is implicitly retrieved (see code example 46).

In addition to these global agent host features offered as a Pini service, an agent host also provides facilities, which are only locally available. They are rather Pini client related mechanisms as it can be derived from the `AgentHostLocal` interface given in code example 43:

```
public interface AgentHostLocal{

  public Agent[] lookupForAgent(ServiceTemplate template, int count);

  public AgentHostGlobal[] lookupForAgentHost(ServiceTemplate template,
                                                          int count);

  public boolean notfiyEvent(ServiceTemplate template, int transition,
                                              EventNotifier notifier);

  public void migrateTo(AgentHostGlobal target, MobileAgent reference)
                                                        throws Exception;

  public Agent createAgent(String aPath, String aConf, Object[] init);

  public void communicate(ServiceID id, Message mes) throws Exception;
}
```

CODE EXAMPLE 43. The `AgentHostLocal` interface

This interface obviously provides SDF mechanisms by the `lookupForXXX(..)` methods, which are directly based on their Pini counterparts. Likewise, the method `notifyEvent(..)` offers the functionality for agents to register for remote events as it is usual in a Pini system. These methods define an agent host as a Pini client. Using this Pini-client related behavior of PLAP agent hosts, PLAP agents obtain the required infrastructure information by accessing the underlying Pini infrastructure and its community management system.

Aside from these SDF-related methods, the local interface defines a `communicate(..)` method allowing a further communication path. However, agent communication facilities will be described later in the section on agents.

More essential is the `migrateTo(..)` method - it is directly invoked by agents in order to initialize the migration process as it was already described in previous paragraphs. This local method is responsible for the preparation and the final migration request to the target host. For example, an agent that wants to migrate must switch from the `active` state to `transient` state, which defines the agent as temporarily not available (for such states please refer to [35] and [66]). This is accompanied by a complete de-registration/removal of the agent from the plug-and-participate system. Moreover, also the reference to the local agent host as well as the (current) agent address must be reset - at the new host

they will be invalid. The preparation is finalized by invoking the `migrateTo(..)` method of the given reference to the target agent host, which will launch the already described procedure:

```
public void migrateTo(AgentHostGlobal target, MobileAgent reference)
                                                        throws Exception{
  //invoke the beforeMove
  reference.beforeMove();
  //get the ID
  ServiceID agentID = reference.getAgentID();
  //prepare the agent
  reference.prepare();
  //de-register the agent
  Lease lease = (Lease)localAgents.get(agentID);
  leaseRenewalManager.cancel(lease);
  //get the remote event registrations
   . . .
   try{
        //initialize the migration
        target.migrateTo(agentID,agent.getConfig(),null,reference);
   }catch(Exception e){. . .}
}
```

CODE EXAMPLE 44. The local migration facility of a PLAP agent host

In addition to this technological view to the migration process, also a specific conceptual feature must be considered: The PLAP agent platform requires that the agent code is pre-installed at each possible and allowed target platform. This rather essential constraint is motivated by several facts:

(1) Security Issues

If an agent were to retrieve its code from its specific codebase, it executes foreign code at the target device. Such code might be corrupt and cause undesired effects (see also [90]). Especially at automation devices this must be avoided - if the code is pre-installed, it can be assumed trustworthy.

(2) Limited Device Related Constraints

In order to run programs stable on limited devices, their resource consumption must be known in advance. If agents were to download their own code, this constraint could no longer be kept - dynamically downloaded code results in an unpredictable memory consumption.

(3) Technological and Platform Related Constraints

Limited devices often do not support dynamic remote class loading due to the already discussed constraints.

Considering these facts, the requirement to pre-install the agent code is reasonable. Moreover, it neither contradicts the mobility paradigm of agents nor the general definition of mobile agents - mobility with respect to agents simply means that specific entities such as objects are able to autonomously

navigate through network infrastructures. PLAP agents follow this definition, since they can freely move throughout the network. They choose their target platforms autonomously depending on their own criteria.

Finally, the `AgentHostLocal` interface defines a `createAgent(..)` method, which is mainly concerned with the Agent Management System. It is the most essential one in the interface, because it defines the creation of agents, which are the major basis of each agent community. Such a creation process of PLAP agents is simple and relies on the parameters passed to the method:

- First, the plain agent object is created using the `Class.forName(..)` statement and subsequently the `newInstance()` method of the created `Class` object. In this process, the passed (class) name of the agent is used in order to load the corresponding class.
- In the second step, the agent must be configured. This means, the evaluation of an agent configuration class comprising basic information for that particular agent as a simple Pini service.
- The third step deals with the initialization of the agent - the `init` parameters from the method invocation are passed to the agent. Establishing the agent in the system as an arbitrary Pini service finalizes the creation process:

```
public Object createAgent(String agentPath, String agentConfig,
                                                    Object[] init){
  try{
     //create the agent object
     Class agentClass = Class.forName(agentPath);
     agent = agentClass.newInstance();
     //initialize the agent
     ((Agent)agent).setConfig(agentConfig);
     ((Agent)agent).init(init);
     //initialize the agent as Pini service and register
     ((DefaultPiniService)agent).init(agentConfig,agent);
     register((Agent)agent);
     //init the agent
     ((Agent)agent).init(init);
     //launch the agent
     ((Agent)agent).main();
  }catch(Exception e){. . .}
  return agent;
}
```

CODE EXAMPLE 45. The creation of PLAP agents

If these steps have been successfully performed, the agent can be started via its `main()` method so that it behaves autonomously in order to fulfill its associated tasks. Agent behavior, however, is the topic of the following section.

7.3.2.2 PLAP Agents

Several basic PLAP agent facilities such as mobility, access to agent hosts and their basic behavior as arbitrary Pini services have already been described in previous sections. These facilities can be classified into basic plug-and-participate related features (which are usually transparent for users) and agent related features as shown in the following abstract class:

```
public abstract class Agent extends DefaultPiniService
                                          implementsAgentCommunication{

  protected AgentHostLocal localHost  = AgentHost.getLocalHost()

  public final void setAgentID(ServiceID id){...}

  public final ServiceID getAgentID(){...}

  public abstract void init(Object[] initial);

  public abstract void communicate(Message mes);

  public abstract boolean alive();

  public abstract void main();
}
```

CODE EXAMPLE 46. The class interface of a basic PLAP agent

Basic plug-and-participate facilities are rather transparent - each PLAP agent is an arbitrary Pini service, and therefore extends the `DefaultPiniService` class. During its creation, the agent is registered with the underlying Pini system. This is triggered by the agent host as it was shown in the previous section. The agent host, likewise, is responsible for maintaining the agent registrations. Further plug-and-participate related facilities such as agent search and participation in the passive information system, are performed explicitly and must be implemented by the developer - these features are abstracted by the PLAP environment so that a developer does not need to deal with Pini-specific features. They are accessed through the PLAP agent host interfaces.

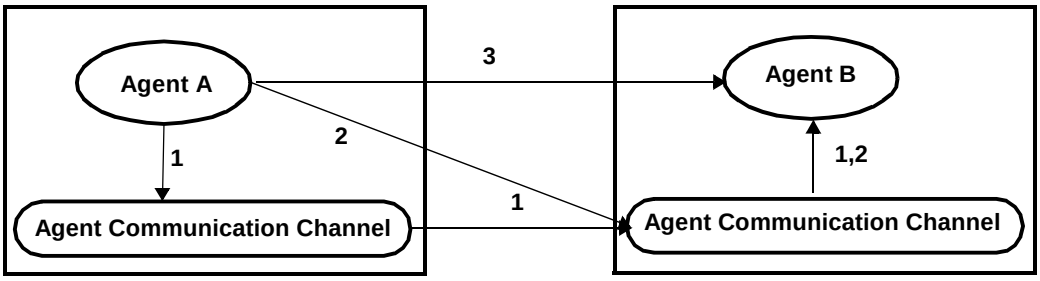
Moving the focus to the agent-related concepts, two particular categories can be classified: communication mechanisms and agent management features. While the `setAgentID(..)` and `getAgentID()` methods are final and thus cannot be changed, the `alive()`, `init()`, and `main()` methods must be implemented by the developer. In particular, the two final methods deal with the identification of agents, and thus must not be manipulated in order to ensure stable and robust communities. The following abstract methods, in contrast, allow the definition of the agent behavior:

- The `init(..)` method is invoked by the PLAP environment after the agent is created. It is used to setup the agent and the customize it.
- The `alive()` method, in turn, is used to determine whether the agent is still active or not.

Remark: This method checks the status of the agent - 6 different states of agents are defined in FIPA: `unknown`, `active`, `suspended`, `waiting`, `transit` and `initiated`. For more details see [35] and [66].

- The `main()` method is the central method of a PLAP agent; it controls the agent. It is automatically invoked by the PLAP environment in order to launch the agent and to start its actions.

The second category of agent-related features is the communication within agent systems. Agents generally receive communication messages via their dedicated `communicate(..)` method. Such messages are usually received remotely, and therefore each agent also implements the `AgentCommunication` interface, which standardizes this behavior (compare with code example 46). This communication facility defines the basic Pini service of an agent - the installed skeleton implements this interface at least. Together with the `communicate(..)` methods of the agent host interfaces, three different communication paths result. They are illustrated in the following figure.



1 - communication via ACC
2 - communication via ACC, the variant through the remote ACC
3 - direct communication between the agents

FIGURE 24. The structure of the communication facilities of PLAP, in principle

For example, communication path 1 relies on the invocation of the `communicate(..)` method defined in the `AgentHostLocal` interface. This local agent host determines the (remote) host where the receiver resides. Invoking the `communicate(..)` method of this corresponding agent host (defined in the `AgentHostGlobal` interface), the message is transferred to the target host where it is finally dispatched via the agent-related `communicate(..)` method. The second alternative (path 2) requires that the agent retrieves the host reference of the target agent - it invokes the `communicate(..)` method of the remote host, which finally delivers the message. Path 3 finally defines the direct communication - the sender retrieves the agent reference of the target agent and invokes its `communicate(..)` method directly.

All these general facilities are provided by all agents, and therefore they are aggregated within the abstract class `Agent`. PLAP, moreover, provides a distinction into mobile agents that require some additional methods, and stationary agents that do not define any further features. This leads to the definition of specific classes with respect to the agent behavior: `MobileAgent` and `StationaryAgent`. Both are abstract classes, too, extending the class `Agent` - `StationaryAgent` does not specify any further methods, while `MobileAgent` does, namely `afterMove()`, `beforeMove()` and `prepare()`, which are directly related to the migration concept. The `prepare()` method is a final

method, and must not be manipulated. In contrast, `afterMove()` and `beforeMove()` are abstract and must be implemented by the developer. Each of the methods is invoked by the PLAP environment either to trigger user-defined operations associated with migration processes, or to prepare an agent for its migration - the invocation of these methods is examplary shown in code examples 42 and 44.

7.3.3. Evaluation of the Concepts Respective to Plant Automation

The PLAP agent system describes a FIPA compliant concept supporting mobility of agents. This covers one facet of automation needs - mobility of agents in automation allows flexible, robust and reliable solutions. PABADIS, for example, offers a completely autarkic machine control mechanism through mobile agents - PAs migrate to the machine and perform their task locally so that no remote control is necessary.

PLAP defines standardized but generic communication paths - in automation and planning systems autonomous agents must also be able to cooperate with each other, and thus they must communicate.

The third aspect concerns the target area of distributed automation applications like PABADIS, namely automation devices. PLAP provides a limited-device suitable agent platform, which allows the application of the automation solutions on automation devices directly. It follows, in this respect, the “agentification” trend in the automation field.

While the first and the second aspects are derived from the objectives and research in the PABADIS project, the third statement must be proven using appropriate measurements, which are topic of the following section.

7.3.4 Evaluation of Resource Consumption

The evaluation of the PLAP resource consumption follows the established patterns of the resource evaluation given in sections 6.4 and 7.2. Consequently, the focus is on memory consumption - particularly disk space, runtime and process memory requirements.

Except for the disk space consumption, the measured values are compared with the prerequisites of automation devices in order to prove that the PLAP concept generally fits to them. The evaluation is divided into three parts - the evaluation of PLAP’s disk space consumption; a general evaluation based on a simple PLAP application consisting of two agent host applications and a simple mobile agent; and an evaluation of the final migration of PABADIS CMUs to automation devices and the KVM/CLDC platform, at least with respect to the facilities of the RA.

PLAP Disk Space Consumption

The PLAP disk space consumption basically covers two aspects - the PLAP framework itself and the underlying Pini framework. These two parts result from PLAP’s design: It essentially relies on Pini features and thus can be assumed to be a rather complex Pini application. The disk space consumption of PLAP must therefore be seen in conjunction with Pini’s consumption. Such a fundamental relation of

PLAP and Pini furthermore justifies PLAP's significantly low disk space requirements given in table 18, because several facilities are directly derived from Pini:

Package	Size on Disk (kB)
`plap.kvm.agent_host`	21.20
`plap.kvm.agent`	2.27
`plap.kvm.util`	1.90
Sum	**25.37**

TABLE 18. The disk space consumption of the PLAP packages

Remark: This measurement relies on a proof-of-concept implementation, which does not provide all facilities. For example, PLAP currently provides agent creation, mobility of agents, management of IDs and leases, and communication. It does not support remote events.

The given measurement relies on the accumulation of the disk space consumption of PLAP packages which are derived from the general infrastructure given in the introduction to this PLAP section, namely agent host, agent and LUS. However, the LUS facility is excluded, since it is provided by the Pini framework. Two PLAP component related packages result, which are extended by a further package providing basic tools and data structures used within the PLAP architecture - the `plap.kvm.util` package.

If the measured 25.37 kB of the plain PLAP framework are added to Pini's size on disk (211.18 kB, see table 8), a total of 236.55 kB results. This obviously fulfills the automation prerequisites listed in table 2, and can be installed on the general limited devices shown in table 4.

Evaluation of a Simple PLAP Application

This simple agent application basically covers all PLAP features, which are also automation relevant: The agent hosts simply wait until a mobile agent visits their host. Both hosts only provide the defined PLAP features required for hosting agents - this ensures objective measurements, which are not falsified by memory consumption of additional mechanisms. This also applies to the mobile agent. It relies on the `MobileAgent` class. The agent structure and its behavior are simple: Once the agent is created and launched, it starts a `Thread` for all its actions. These actions are limited and directly related to the migration - the agent searches for available agent hosts and simply migrates there. This behavior results in a ping-pong effect, which obviously incorporates all basic PLAP features such as local and global agent host facilities as well as mobility, while communication and remote event concepts have been neglected. The simplicity of the entire application can also be derived from its disk space consumption listed in table 19:

Entity	Size on Disk (kB)
Host 1	0.42
Host 2	0.57
Agent	6.07
Sum	7.06

TABLE 19. Disk space consumption of a simple PLAP example

The 7.06 kB required is rather low; this gives a first impression of the general PLAP resource consumption and shows that the concept smoothly fits to limited devices, especially devices within the automation field.

Remark: The difference between Host 1 and Host 2 results from the initialization of the agent at Host 2 - this requires a few additional code statements.

The last statement - that PLAP fits the automation device prerequisites - can be validated by measuring the memory consumption regarding runtime and process memory:

Entity	Runtime Memory (kB)	Peak (kB)	Process Memory (kB)
Host 1	~121	~136	~800
Host 2	~116	~131	~800

TABLE 20. Average values of runtime memory consumption, peak values and process memory

This table basically comprises the same categories as in previous measurements, but is extended by a peak value column. These peaks values indicate the highest memory consumption of the application.

They can usually be observed, if the agent migrates to the host and resides there, while on the source host the required memory for the agent is released - figure 25 illustrates this behavior:

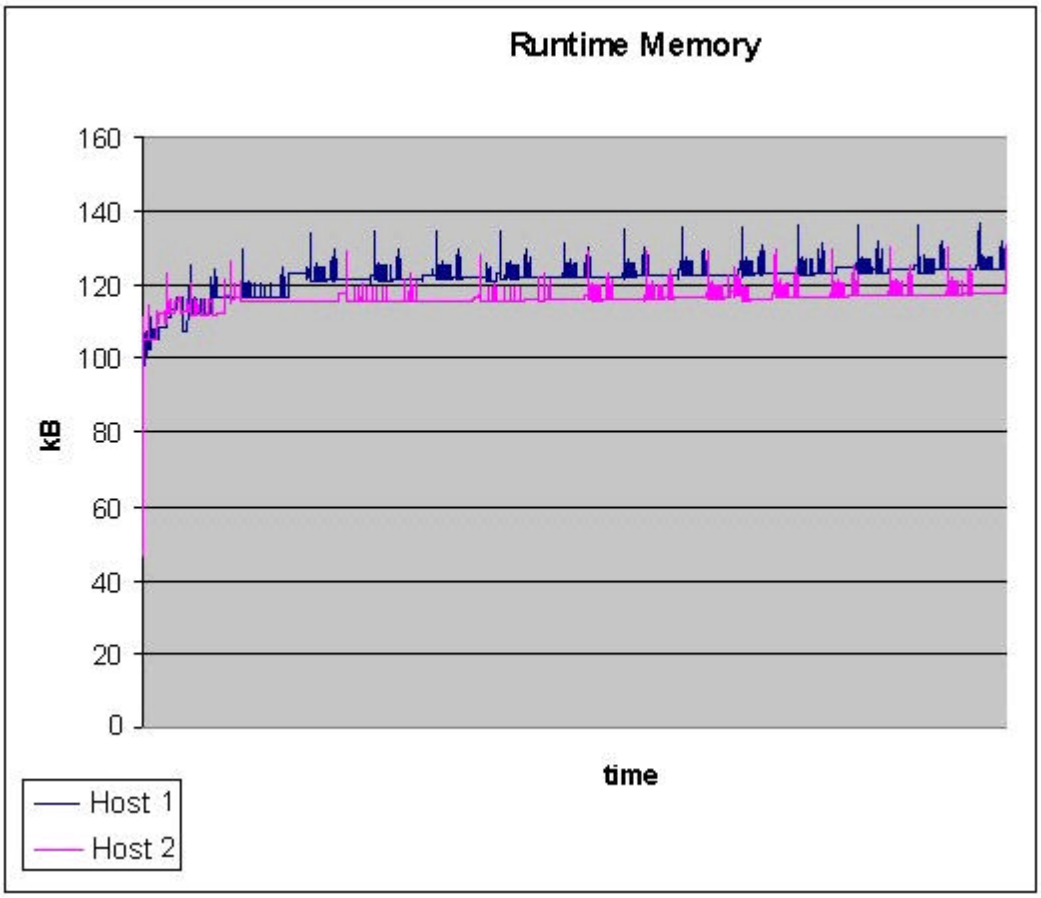


FIGURE 25. The memory consumption of the PLAP example illustrating the migration of agents

This figure also validates the measurements given in table 20, and stresses the mobility of agents in terms of memory consumption: When the agent resides on a particular host, the consumption is increased by about 15 kB, while the memory consumption of the source host is obviously decreased - a kind of base line can be derived. The two resulting base lines of both hosts approximately converge against the calculated average value, while the peaks, shown in figure 25, illustrate the measured values.

Comparing these runtime memory values with the prerequisites of automation devices listed in table 2 underlines the expected result - PLAP fits the memory prerequisites of automation devices in all cases, namely average and peak runtime memory requirements as well as process memory consumption.

However, these observations only consider a simple application that is not directly related to automation. PLAP as a basic technology for plant automation is the topic of the following section covering the final migration of PABADIS CMUs to the KVM/CLDC platform, and therefore to automation devices directly:

A PLAP-based PABADIS CMU

The third and final part of this PLAP evaluation concerns the migration of the PABADIS CMU implementation to the PLAP agent system. This migration focuses on the adaptation of the RA to PLAP without considering the PABADIS agent abstraction. It relies on the Pini-based CMU variant discussed in section 7.2. For purposes of demonstration, the agent abstraction can reasonably be neglected in order to allow an objective comparison of both variants: the KVM/CLDC variant from section 7.2 without PLAP features and the version migrated to PLAP.

The KVM/CLDC compliant RA version resulting from section 7.2 simply extends the abstract PLAP `StationaryAgent` class and implements the defined methods. This increases its size on disk:

Entity	Size on Disk (kB)
RA simulated	35.7
RA using PLAP	45.2

TABLE 21. Comparison of the disk space consumption of a simulated RA and a PLAP based RA

Moreover, in order to run the CMU and the RA, an agent host module must be initialized, which is done in the `CMU` class - only a few statements need to be added to the class. This results in the final disk space consumption of the entire CMU:

Entity	Size on Disk (kB)
CMU at all without PLAP	177
CMU at all using PLAP	187

TABLE 22. Comparison of the disk space consumption of an PLAP-based CMU

The measured values obviously fulfill the prerequisites of automation devices, and thus the PABADIS CMU can be installed on such devices. Moreover, also the consumed runtime and process memory fit the automation prerequisites as table 23 shows:

Entity	Runtime Memory (kB)	Peak Value (kB)	Process Memory (kB)
PLAP-based CMU	~ 220	~242	1020
PABADIS reference implementation	~1218	--	22000

TABLE 23. Pini/PLAP based CMUs versus PABADIS reference implementation

Remark: The measured runtime values slightly differ from those given in table 15. This decreased consumption results from Java's unpredictable garbage collection. It removed non-referred objects so that a decreased runtime consumption results despite increased functionality. However, the magnitude of the consumption is equal to that given in table 15.

Comparing this measurement with the memory prerequisites of automation devices finally proves PLAP, Pini and PABADIS as innovative limited-device compliant solutions. In contrast, the current PABADIS reference implementation relying on Jini and Grasshopper consumes 20 times more process memory than the limited-device compliant variant. Likewise, the runtime consumption differ - the PLAP/Pini-based CMU variant requires only one fifth of the reference implementation.

Remark: Such a comparison between the PABADIS reference implementation and the PLAP/Pini-based CMU must be seen on an abstract level, because the limited-device variant offers fewer features than the PABADIS reference implementation. However, the comparison gives an impression of the ratios between limited device suitable variants and their counterparts.

An important aspect in modern plant automation systems is the provision of integrated solutions covering all levels shown in the automation pyramid. With PABADIS, an automation solution is provided that integrates the MES level and the field level as well as connects to the ERP level. This PABADIS framework is fundamentally based on two genres of technologies offering solutions for distributed systems, namely agent and plug-and-participate technologies. Within this thesis two particular representative technologies are provided, which finally allow the application of (at least) PABADIS to automation devices directly - an innovation in this field. Such a migration is based on the innovative features of the provided technologies Pini and PLAP.

However, Pini and PLAP define a migration step towards automation systems that covers only one direction; a connection to upper levels is also required in order to complete the integration. This facility will be implemented by a dedicated gateway, which allows the connection between limited device environments and the office world - the gateway connects Jini and Pini applications seamlessly.

7.4 JPGateway - A Gateway between Jini and Pini

Integrated automation solutions require a seamless connection from the office level down to the field level and vice versa. This integration covers several aspects such as an harmonized physical connection, e.g., via Ethernet as proposed by PABADIS and also IAONA (see the IAONA home page - http://www.iaona.org), and also software- and application-related aspects. Limiting the focus to software and plug-and-participate-based applications, an integration is required, which allows the transparent use of software/application components on each level such as the use of services in automation and at the office level regardless of their technology of origin. An integration like this is provided by the JPGateway, which transparently connects Pini and Jini - Jini applications can be used as Pini applications on limited devices, and vice-versa (see [21, 23]). This leads to two different integration directions, which are always driven by the facilities of both technologies. Both directions will be shown in separate sections describing the easiness of connecting these two stand-alone plug-and-participate technologies. The major aspect in this connection is the use of different platforms: Jini requires a Java 2 Standard Edition environment, or at least a Java 2 Micro Edition CVM/CDC environment, while Pini relies on the KVM/CLDC (J2ME) platform. These different platforms, moreover, justify one design decision: The JPGateway must run on a non-limited-device platform like the Java 2 Standard Edition, because it must participate in both technologies simultaneously in order to retrieve and provide services.

7.4.1 Jini-to-Pini

In order to offer Jini services in the Pini world, the gateway behaves as a Jini client, while in Pini it offers a corresponding Pini service. Providing this Pini service requires the initialization of a related skeleton and the `PiniRMI` infrastructure. The skeleton reflects the facilities to be provided, which finally reflects the gateway behavior as a Jini client: It requests the service proxy from the Jini architecture and introspects the proxy regarding its provided methods. Based on this knowledge, a so-called gateway intermediate service is created as well as the corresponding `PiniRMI` skeleton. This gateway intermediate service represents a Pini service - it installs the skeleton, and is registered with the Pini-LUS architecture. The required code for this purpose is created dynamically - the skeleton is generated by the `PiniRMI` compiler, while the corresponding service code results from a further module that generates a specific class. This class maps the received method call requests to Jini service proxy methods. All shown components together form an architecture that behaves as a Jini client and a Pini service simultaneously. Its general control flow can be described by an arbitrary service use request: If a Pini client accesses the service via Pini's service proxy stub concept, the gateway intermediate service receives the request and forwards it via the Jini proxy to the Jini service. Figure 26 illustrates this simple gateway architecture, which offers Jini services in the Pini world:

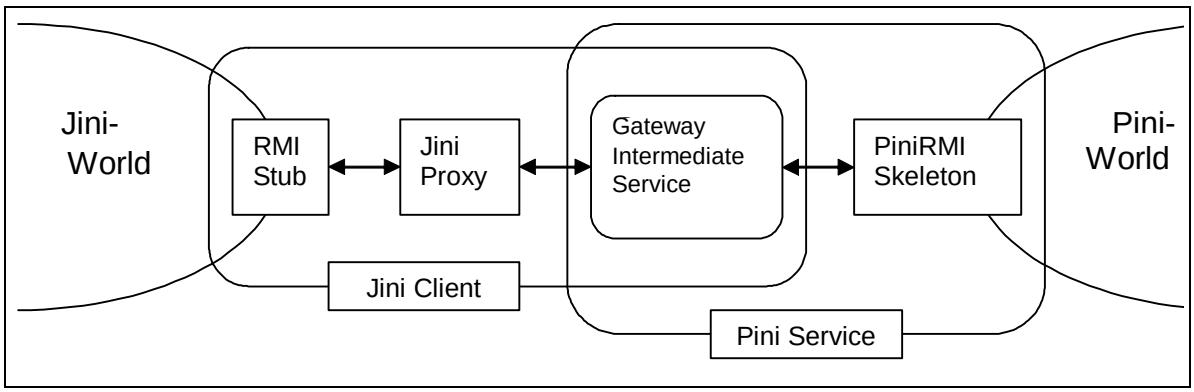


FIGURE 26. General structure of the Jini-to-Pini direction of the JPGateway

A second approach in offering Jini services in the Pini world that has the same architecture but relies on a direct mapping without class code generation likewise involves the evaluation of the Jini proxy, the creation of a method table of this proxy, and the initialization of a dedicated Pini service architecture - this service architecture comprises a special skeleton that receives the `MethodCallEvents` from its associated `MethodCallListener` as described in section 6.2.1.2. The received method call requests and their parameters are mapped to the relevant method in the table, and the method is finally invoked via appropriate Java reflection mechanisms. The gateway intermediate service therefore joins the Pini architecture in the same fashion as defined for the corresponding service interface, but it simply performs a direct mappping of requests without any code generation.

Both approaches, however, rely on a specific prerequisite: Each object to be transmitted as a method invocation parameter must be `PiniSerializable`in order to be compliant with the Pini architecture.

Moreover, the JPGateway requires all classes of the parameter or return value objects in order to be able to restore and dispatch the objects to their receivers.

7.4.2 Pini-to-Jini

This migration direction generally works in the same fashion as the direction from Jini to Pini - the gateway behaves as a Pini client and offers an equivalent Jini service: It initializes a Pini-related service proxy stub, which forwards requests from Jini clients to Pini services. The gateway intermediate service receives requests from Jini clients, which defines it as a Jini service. Such requests are mapped to Pini compliant method invocations and forwarded to the installed Pini service proxy stub - the gateway intermediate service simultaneously behaves as a Pini client. Being a Jini service requires a registration with the Jini LUSs and the provision of a Jini proxy - in order to reduce the effort, this code generation is simply based on the interfaces of the Pini service. The resulting classes represent a Jini service as a Java remote object, and a service proxy, which connects to its service via Java RMI.

Initializing the Pini proxy stub, however, requires the class code of the proxy stub, which is usually not provided as a downloadable class in Pini - a Pini service that wants to be published to the Jini world must offer its service interfaces for download. Using this facility, the corresponding proxy stub can be generated at the gateway. The interfaces are provided by a HTTP server, and additionally the codebase is published in the corresponding `ServiceDescription` object of the service.

Finally, an architecture results in which the gateway intermediate service behaves as a Jini service (it represents a Java remote object) offering a service proxy, while this service simply maps requests from Jini clients to the corresponding service proxy stub:

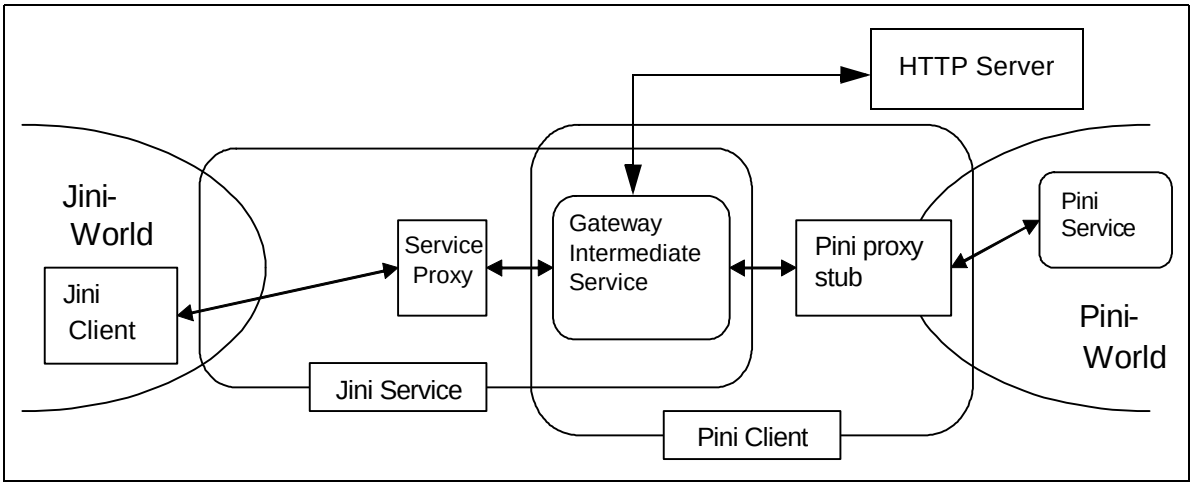


FIGURE 27. General structure of the Pini-to-Jini direction of the JPGateway

However, this integration direction, as in Jini-to-Pini, requires that each parameter and return value is `PiniSerializable`. Likewise, all required classes must be accessible to the JPGateway framework. Otherwise, the objects cannot be restored and therefore are not dispatched to the corresponding recipient of a method invocation.

CHAPTER 8

Conclusion and Outlook

The thesis will be recapitulated by checking the results of this work against the motivation and the dedicated goals set there. Likewise, an outlook is given showing possible future work and fields.

"A successful manufacturing business is one which is able to respond rapidly to changes." (see [13], chapter 3, p. 5) - this is the motivational phrase of this work and sketches the main ideas of the entire thesis. Such an ability to rapidly react to changes requires dedicated re-configurability features of a system, which can be provided by plug-and-participate technologies. An example of a revolutionary automation solution relying on distributed systems was investigated in this thesis - PABADIS - to which the author made a major contribution regarding the required re-configurability of systems. The defined concept shows a major impact on planning and production control - the PABADIS solution relies on plug-and-participate technologies as one major basis, and agent technologies as the second. Both technology genres cope with the two covered integration directions of PABADIS, which ensure this kind of re-configurability:

- **Horizontal integration**: Using plug-and-participate technologies ensures highly flexible and adaptable infrastructures of machines and their functions. Such networks provide abstracted functions, and are established spontaneously.

- **Vertical integration**: This direction is realized via agent technologies allowing an autonomous planning of the tasks contained in manufacturing orders. The tasks are finally dispatched to machine functions provided by plug-and-participate technologies - all levels of the automation pyramid are coverd, and thus a completely integrated solution is provided.

Both integration directions of PABADIS are closely related to each other - agents performing the planning require information about the underlying function infrastructure provided by a plug-and-participate concept. Moreover, agents control the manufacture of their products, and therefore must access the provided functions of machines. PABADIS abstracts these functions, and provides them via plug-and-participate facilities to agents. PABADIS defines CMUs, which offer mechanisms supporting the agents in performing their tasks. This thesis especially covered the function provision part via plug-and-participate, as it was the author's specific research contribution in this field: Three dedicated facets were investigated ensuring efficient function provision, namely plug-and-participate, (especially in) automation solutions and limited devices.

PABADIS is an example that combines all three facets, since it defines a plug-and-participate based automation solution, which can be applied to automation devices directly. This is a fundamental advantage, because legacy automation devices are characterizable as limited devices and usually provide limited device platforms. Such a limited device character has been outlined by an evaluation of the state of the art in automation. Based on this preliminary evaluation, an alleviated classification of the results of this thesis regarding state-of-the-art automation solutions and technologies is possible. For example, in the field of plug-and-participate technologies, the classical representatives - Jini and UPnP - were investigated regarding their capabilities, features and requirements. Both are embedded into the middleware field, which was also briefly described. Likewise, the automation field was surveyed - proposals like HMS, IDA and PROFInet were investigated in order to gain an overview of the basic objective of distributed automation. It ends in an evaluation of the target platforms of such solutions, namely legacy automation devices. This evaluation characterized these devices as limited devices.

Motivated by the information about legacy technologies and automation solutions, but mainly through the need for flexible and adaptable automation environments, the PABADIS concept was introduced. After an overview of basic concepts and ideas, the focus was concentrated on one major aspect of PABADIS: generic function provision via plug-and-participate facilities. Under consideration of the necessities of the automation field regarding plug-and-participate, an abstract interface was developed (see figure 15), which furthermore relies on general facilities of plug-and-participate technologies. These features are abstracted and aggregated in a corresponding abstract plug-and-participate interface given in section 2.3.1. The interface mentioned first is finally the basis for the PABADIS plug-and-participate module shown in chapter 5 of this thesis - as one specific contribution of this thesis to the field of automation. Moreover, it is completely independent from any plug-and-participate technology, and therefore provides an abstraction layer for PABADIS. Based on this abstraction layer, PABADIS CMUs offer their functions in a homogeneous fashion to the community - this ensures the horizontal integration. Agents, in turn, access these functions in always the same way via FCM and ECM, allowing a generic access to machines and furthermore an efficient machine control. This concept is revolutionary in this field and allows innovative solutions. In particular, these automation related plug-and-participate concepts are one main result of the research effort done by the author - the solutions have been presented in several articles on conferences and in journals, as well as presented to industrial partners.

However, if such solutions are to be applied to automation devices directly, appropriate limited device constraints must be considered: The underlying agent and plug-and-participate technologies must comply with these constraints. This led to the development of the Pini plug-and-participate concept, which relies on the chosen automation reference platform KVM/CLDC: Pini implements the abstract plug-and-participate interface defined in section 2.3.1, and furthermore relies on an interface suite similar to Jini - this design decision was justified by the easiness of migration, alleviated comparison of concepts, powerful facilities relying on the interface suite, and the final compliance with the abstract plug-and-participate interface. Likewise, the claim of Pini as a limited Jini has been invalidated due to Pini's completely different ideas and infrastructures. This is achieved by innovative concepts of Pini, namely the `ServiceDescription` facility, the service use concept and the distributed Pini-LUS architecture as a community management system. Moreover, Pini allows the exact determination of the memory consumption of provided services in advance, which is the crucial point in this field and especially in automation with its strong realtime constraints - classical plug-and-participate technologies cannot provide such essential facilities. Pini is further related to automation devices/automation environments, since it relies on a network abstraction layer, which allows Pini to cope transparently with most of the relevant communication technologies. This automation compliance of Pini is finally proven by a Pini-based PABADIS CMU implementation, which was compared with the Jini-based counterpart. The obvious result of this comparison is Pini's compliance with the automation field in terms of less memory consumption, its compliance with the provided platforms and a small footprint in general. The crucial point is the possibility to estimate the memory consumption of services in advance. In contrast, the Jini-based reference implementation suffers from disproportional resource consumption and platform requirements.

A second aspect with respect to Pini's applicability to automation devices is the provision of the Pini-based Lightweight Agent Platform (PLAP), which even provides mobility of agents in resource-constrained environments. It therefore follows the trend of "agentification" in the automation field. PLAP essentially relies on Pini and uses its basic facilities - the distributed Pini-LUS defines an efficient agent search component, while agents and agent hosts are basic Pini services. The relation to Pini shows several important benefits with respect to automation:

- PLAP has a very a small footprint in terms of the pure PLAP application.
- Automation solutions like PABADIS can save disk space, if two different technologies, namely plug-and-participate and agent technologies, are available, which rely on the same basic framework.
- PLAP relies on the KVM/CLDC platform and the most important facilities are derived from Pini.
- PLAP consumes very few resources due to the resource-efficient Pini architecture.

PLAP allows a complete migration of PABADIS to automation devices directly. As an example, the CMU concept was migrated to PLAP/Pini, namely the RA was adapted to a PLAP-based agent. Finally, measurements of memory consumption have shown that PABADIS is a suitable limited device automation solution. Due to Pini and PLAP and their innovative facilities, PABADIS can be applied to automation devices directly; this was proven by measurements of the resource consumption of Pini- and Jini-based PABADIS CMU implementations. Considering the process memory consumption, a ratio of 1:20 results, which indicates a significant advantage of Pini/PLAP-based CMUs against Jini/ Grasshopper-based counterparts.

This migration to automation devices directly refers to a further fact: the added value and the efficiency of controllers can be increased. Moreover, if the controllers are equipped with such distributed automation solutions such as PABADIS, their “intelligence” can be utilized in a more efficient and effective way, which also increases the added value of the controllers.

Likewise, the efficiency of machines can be increased - machines usually consist of many controllers, and if they provide more facilities, the added value of the machines increases. This also tackles the costs of machines as a driving factor - if automation applications can directly be applied, money can be saved; otherwise, suitable additional equipment must be obtained increasing the financial investment, and simultaneously decreasing the efficiency of such machines. Pini and PLAP finally allow the application of (distributed) automation on limited automation devices directly, and therefore their concepts and ideas can be assumed as a major fundament for modern distributed automation.

Although PABADIS provides an integrated solution covering all levels of automation from ERP via MES down to the field level, it mainly concentrates on MES related tasks. In order to provide solutions with flexibility and adaptability spanning the entire automation field, also the remaining fields must be investigated with respect to innovative solutions and concepts. This thesis offered two specific examples for the field level - the semantic fieldbus approach allows an alleviated setup and maintenance of such networks and infrastructures, and the concept for totally distributed automation generally concerns the definition of automation applications. In both fields, further research efforts are necessary, especially the definition of appropriate ontologies for automation or fieldbus environments. Likewise description facilities for fine grained function provision rely on the ontology concept - both aspects are covered in a project proposal to the EC named CODING. For example, in CODING the automation-related plug-and-participate interface as well as the concept of FCM/ECM for function control will be used as a basis for the concepts to be developed.

Further research effort is required with respect to the extensions of IDA and PROFInet as shown in section 5.4.2. First, the concept must be defined in more detail, and must be harmonized with the original solutions. Second, a reference implementation must be provided. Based on such a reference implementation, the concept must be evaluated in real automation applications - this, e.g., also concerns the sematic concepts as a crucial point for the proposed mapping facilities.

These rather field level related concepts must be embedded into a broader context, namely integrated automation solutions are required, which cover almost all fields of automation. This means, these field level solutions must be part of a vertical integration, e.g., as proposed by PABADIS’ agent concept. In particular, such an integration requires intensive research in order to provide next generation automation solutions that are always able to cope with future market situations. This necessary research likewise requires permanent investigations on MES level in order to satify customer demands and to survive in the markets.

In addition to this rather research-related work to be done in the future, also the PABADIS concept needs to be further evaluated, and finally applied to real plants. This basically means the definition of pilot applications, in the best case the design of a plant infrastructure for industrial purposes like car manufacturing. For example, appropriate steps are to be performed in order to equip a legacy machine and/or a plant with a PABADIS system:

- adaptation of the automation function with respect to the FCM/ECM protocols
- installation of the generic CMU software

- preparation of PABADIS conform Capability Descriptions and Work Orders
- integration of CMUs into a broader system
- preparation of the connection to the ERP system.

These steps, especially the migration of PABADIS software to machines furthermore requires a (partial) re-design of the software with respect to the constraints of limited devices and the target controller hardware: An example for such a migration was given in section 7.2, where a CMU has been migrated to Pini and finally to Pini/PLAP (section 7.3).

In order to further optimize the concept, also a re-design of the Pini- as well as PLAP-implementation is recommended - the given reference implementations are basic proofs of the concepts, and thus a software re-design will safe further resources, increase the efficiency of the concepts, may eliminate implementation failures, and may even improve the concepts themselves. Such an improval of the concepts usually results from further experiments and measurements, which must be done based on the more optimal implementation - a further field of future work can be identified as the execution of new measurements covering further aspects of the technologies and concepts.

Finally, the three facets of this thesis have been detailed in Pini, PLAP and the PABADIS approach. The resulting solution provides the basic requirements regarding the statement, which opened this thesis as well as this conclusion - the statement will be modified with respect to the results of this thesis: "Pini and PLAP are the basis for PABADIS as an automation solution for sucessful manufacturing business which is able to respond rapidly to changes."

Abbreviations

ACC	Agent Communication Channel
AF	Automation Function
AMS	Agent Management System
API	Application Programming Interface
ARP	Address Resolution Protocol
CD	Capability Description
CDC	Connected Device Configuration
CFM	Common Feature Module
CIM	Computer Intergrated Manufacturing
CLDC	Connected Limited Device Configuration
CMU	Cooperative Manufacturing Unit
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CVM	Consumer Virtual Machine
DCOM	Distributed Component Object Model
DHCP	Dynamic Host Configuration Protocol
ECM	Event Control Module
ERP	Enterprise Resource Planning
ES	Engineering System
FCM	Function Control Module
FIPA	Foundation for Intelligent Physical Agents
GENA	General Event Notification Architecture
GIOP	General Inter-ORB Protocol
HMI	Human Machine Interface
HMS	Holonic Manufacturing Systems
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IDA	Interface for Distributed Automation
IDL	Interface Definition Language

IIOP	Internet Inter-ORB Protocol
IMS	Integrated Manufacturing Systems
I/O	Input/Output
IP	Internet Protocol
IPC	Industrial PC
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JDK	Java Developer Kit
JRMP	Java Remote Method Protocol
JVM	Java Virtual Machine
KVM	Kilo Virtual Machine
LEAP	Lightweight Extensible Agent Platform
LUS	Lookup Service
MES	Manufacturing Execution System
MO	Manufacturing Order
ORB	Object Request Broker
OS	Operating System
PABADIS	Plant Automation BAsed on DIstributed Systems
PA	Product Agent
PC	Personal Computer
PLAP	Pini-Based Lightweight Agent Platform
PMA	Plant Management Agent
PLC	Programmable Logical Controller
RA	Residential Agent
RAFI	Residential Agent to Function Interface
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RT	Runtime
RTOS	Realtime Operating System
RTSJ	Realtime Specification for Java
SCADA	Supervisory Control and Data Acquisition
SDF	Standard Directory Facilitator
SDK	Software Developer Kit
SOAP	Simple Object Access Protocol
SSDP	Simple Service Discovery Protocol
TCP	Transmission Control Protocol

TV	Television
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URL	Uniform Resource Locator
UUID	Uniform Unique Identifier
WO	Work Order
WWW	World Wide Web
VM	Virtual Machine
XML	eXtensible Markup Language

References

[1] aJile Systems, Hewlett-Packard, IBM, Mitre, Mitsubishi Electric Corp., Siemens, Sun Microsystems, (1999): "Industrial Automation Extension"; JSR 007;
http://jcp.org/en/jsr/all

[2] American National Standard, (2001): American National Standard for Telecommunications - Telecom Glossary 2000, Plug and Play Definition, ANS T1.523-2001;
http://www.atis.org/tg2k/

[3] K. Arnold, B. O'Sullivan, B. Scheifler, J. Waldo, A. Wollrath, (1999): "The Jini™ Specification";
Addison-Wesley Pub Co; 1 edition (June 1999), ISBN: 0201616343

[4] R. W. Atherton, (1998): "Moving Java to the Factory"; Sun Microsystems;
wysiwyg://116/http://www.insitute.ieee.org/select/1298/java.html

[5] F.X. Bea, E. Dichtl, M. Schweitzer, (1992): "Allgemeine Betriebswirtschaftslehre";
6. Auflage; Gustav Fischer Verlag, ISBN:3-8252-1081-2

[6] T. Berners-Lee, J. Hendler, O. Lassila: "The Semantic Web",
http://www.sciam.com/2001/0501issue/0501berners-lee.html

[7] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer, (2000): "Simple Object Access Protocol (SOAP) 1.1"; From: W3C
http://www.w3.org/TR/SOAP/

[8] J. Brehm, M. Fuchs, (2001): “First Contact: “Bluetooth™ Wireless Technology meets CAN Fieldbus“, APC Pervasive Ubiquitous Computing 2001, München, pp. 19 - 26; ISBN 3-8007-2636-X

[9] J. Broekstra, M. Klein, S. Decker, D. Fensel, I. Horrocks; "Adding formal semantics to the Web"; http://www.ontoknowledge.org/oil/papers.shtml

[10] H. Van Brussel, (1994): "Holonic Manufacturing Systems The Vision Matching the Problem"
First European Conference on Holonic Manufacturing Systems, Hannover, Germany, December 1, 1994

[11] S. Bussmann, K. Schild, (2001): "An Agent-based Approach to Control of Flexible Production Systems"; 8th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Nice, France, pp. 481 - 488; ISBN 0-7803-7241-7

[12] S. Bussmann, (1998): "An Agent-Oriented Architecture for Holonic Manufacturing Control" First Open Workshop IMS Europe, Lausanne, Switzerland, 1998 organized by ESPRIT Working Group on IMS

[13] S. Bussmann, (1999): "Rationales for Holonic Manufacturing Control"; Proc. of the 2nd Int. WS on Intelligent Manufacturing Systems, pp. 177 - 184, Sept. 1999, Leuven, Belgium

[14] A. Chaffee, B. Martin, (1999): "Introduction to CORBA About This Short Course" http://developer.java.sun.com/developer/onlineTraining/corba/

[15] H. L. Chen, (1999): “Developing a Dynamic Distributed Intelligent Agent Framework Based on the Jini Architecture“; Master Thesis

[16] J. H. Christensen, (1994): "Holonic Manufacturing Systems: Initial Architecture and Standards Direction"; First European Conference on Holonic Manufacturing Systems, Hannover, Germany, December 1

[17] J. Cohen, S. Aggarwal, Y.Y. Goland, (1999): “General Event Notification Architecture Base: Client to Arbiter”; http://www.globecom.net/ietf/draft/draft-cohen-gena-client-00.html

[18] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, (2000): "Universal Serial Bus Specification" http://www.usb.org/developers/docs/

[19] S. Decker, F. van Harmelen, J. Broekstra, M. Erdmann, D. Fensel, I. Horrocks, M. Klein, S. Melnik; "The Semantic Web - on the respective Roles of XML and RDF", http://www.onto-knowledge.org/oil/papers.shtml

[20] S. Deter, (2000): "Multicast Routing - Protokolle und Algorithmen" Diploma Thesis, University of Marburg

[21] S. Deter, (2001): "JP-Gateway - Plug & Play für "limited devices"; APC Pervasive Ubiquitous Computing 2001, München, pp. 35 - 44; ISBN 3-8007-2636-X

[22] S. Deter, (2001): "Plug&Participate for Limited Devices in the Field of Plant Automation"; 8th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Nice, France, pp. 273 - 278; ISBN 0-7803-7241-7

[23] S. Deter, (2001): "Plug&Participate für "limited devices" mittels Gateway-Strukturen"; NetObjectDays 2001 Konferenz, Erfurt, Germany, pp. 130-137; ISBN 3-00-008419-3

[24] S. Deter, (2002): "Fieldbus Device Description using Tag-based Trees"; 6th AFRICON Conference IEEE, George, South Africa, pp. 263 - 268; ISBN 0-7803-7570-X

[25] S. Deter, R. Blume, E. Klemm, (2002): "Generic Machine Representation in the PABADIS Community"; E-Business and E-Work Conference 2002 Vol. 2, Prague, Czech Republic, pp. 1134 - 1140; ISBN 1 58603 284 4

[26] S. Deter, B. Krzensk, T. Sauter, (2002); "Plug-and-Play for Semantic Fieldbus Networks"; IEEE International Workshop on Factory Communication Systems (WFCS), Västerås, 28.-30. Aug. 2002, Work-in-Progress Proceedings, MRTC Report No. 61, August 2002.

[27] S. Deter, A. Lüder, J. Peschke, (2002): "XML und Plug-and-Participate zur Flexibilitätserhöhung in der Fabrikautomatisierung"; Tagungsband SPS/IPC/Drives Kongress 2002, pp. 275 - 283; ISBN 3-7785-2863-7

[28] S. Deter; K. Sohr, (2001): "Pini - A Jini-like Plug&Play Technology for the KVM/CLDC", IICS 2001, LNCS 2060, pp. 53-66, 2001; ISBN 3-540-42275-7

[29] E. Dichtl, (1994): "Strategische Optionen im Marketing"; 3. Auflage
Deutscher Taschenbuchverlag, ISBN: 3-423-05821-8

[30] W. K. Edwards, (2000): "Core Jini (2nd Edition)";
Prentice Hall PTR; 2nd edition (December 28, 2000); ISBN: 0130894087

[31] ESMERTEC, (2003): "Jeode Java Runtime Environment"
http://www.esmertec.com/products/products_jeode.shtm

[32] ESMERTEC, (2003): "Jeode Platform"
http://www.esmertec.com/products/jeode_tech.shtm

[33] W. Feibel, (1995): "Complete Encyclopedia of Networking", Middleware Definition, Novell Press, San Jose; ISBN 0-7821-1290-0

[34] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, (1999): "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616
http://www.ietf.org/rfc.html

[35] FIPA, home page; http://www.fipa.org

[36] firstperson Inc., (1994): "Oak Language Specification", v0.2
http://java.sun.com/people/jag/green/OakSpec0.2.ps

[37] E. Gamma, R. Helm, R. Johnson, J. Vlissides, (1994): "Design Patterns Elements of Reusable Object-Oriented Software"; Addison Wesley Professional Computing Series, October 1994, ISBN 0-201-63361-2

[38] Y. Y. Goland, T. Cai, P. Leach, Y. Gu, (1999): "Simple Service Discovery Protocol/1.0"
http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt

[39] J. Gosling, J. Bill., G. Steele, G. Bracha, (2000): “The Java™ Language Specification Second Edition”; Addison-Wesley; ISBN 0-201-31008-2

[40] Handelsgesetzbuch HGB, (2003); DTV-Beck; 40., überarb. Aufl.; ISBN 3423050020

[41] M. Höpf: "Holonic Manufacturing Systems (HMS) The Basic Concept and a Report of IMS Test Case 5"
http://hms.ifw.uni-hannover.de/public/Feasibil/holo2.htm

[42] M. Horstmann, M. Kirtland, (1997): "DCOM Architecture"
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp

[43] P. Huang, V. Lenders, P. Mining, M. Widmer, (2002): "Jini for Ubiquitous Devices"; ETH TIK-Technical Report Nr. 137;
http://www.tik.ee.ethz.ch/~lenders/publication/jini_ubiquitous.pdf

[44] P. Huang, V. Lenders, P. Minnig, M. Widmer, (2002): "Mini: A Minimal Platform Comparable to Jini for Ubiquitous Computing"; International Symposium on Distributed Objects and Applications (DOA), Irvine

[45] IDA Group e.V., (2002): "Interface for Distributed Automation Architecture Description and Specification";
http://www.ida-group.org

[46] Intel Corporation, Microsoft Corporation, (1994): "Plug and Play ISA Specification", v1.0a;
http://www.microsoft.com/hwdev/tech/pnp/default.asp

[47] Intel Corporation, Microsoft Corporation, (1999): "Legacy Plug and Play Guidelines"
http://www.microsoft.com/hwdev/tech/pnp/default.asp

[48] W. Kastner, M. Leupold, (2001): "Discovering Internet Services: Integrating Intelligent Home Automation Systems to Plug and Play Networks", IICS 2001, LNCS 2060, pp. 67-78, 2001; ISBN 3-540-42275-7

[49] W. Kastner, M. Leupold, (2001): "How Dynamic Networks Work: A Short Tutorial on Spontaneous Networks"; 8th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Nice, France, pp. 2295 - 303; ISBN 0-7803-7241-7

[50] A. Klostermeyer, K. Lorentz, (2000): "Paradigmenwechsel in der Fabrikautomatisierung";
in: K. Bender, G. Brandenburg, R. D. Schraft, (Hrsg.), "Tagungsband zur SPS/IPC/Drives/ Elektrische Automatisierung - Systeme und Komponenten", Nürnberg

[51] A. Koestler, (1967): “The Ghost in the Machine”; London; Hutchinson & Co

[52] A. Lüder, J. Peschke, S. Deter, A. Bratoukhine, (2002): "Interaction of Software Agents and Real-Time Industrial Control System within a PABADIS System", 14th Euromicro International Conference on Real-Time Systems, Work in Progress Session, Vienna, Austria, Proceedings, pp. 21-25.

[53] A. Lüder, J. Peschke, T. Sauter, S. Deter, D. Diep, (2003): "Distributed intelligence for plant automation based on multi-agent systems - the PABADIS approach"; to appear in Special Issue on Application of Multiagent Systems to PP&C, Journal of Production Planning and Control, 2003

[54] MES Association, (1997):
http://www.mesa.org

[55] Microsoft Corporation, (1999): "Plug and Play Design Specification for IEEE 1394";
http://www.microsoft.com/hwdev/tech/pnp/default.asp

[56] Microsoft Corporation, (2000): "Universal Plug and Play Device Architecture";
http://www.upnp.org/download/UPnPDA10_20000613.htm

[57] Microsoft Corporation, Digital Equipment Corporation, (1994): "The Component Object Model Specification", v0.95;
http://www.microsoft.com/com/resources/comdocs.asp

[58] Microsoft Corporation, Digital Equipment Corporation, (1998) "The Distributed Component Object Model Specification", v1.0
http://www.microsoft.com/com/resources/comdocs.asp

[59] NewMonics, (1998): "NewMonics Offers New PERC Integration Kit to pSOS Users"
http://www.newmonics.com/news/press10.shtml

[60] Object Management Group: “IIOP: OMG's Internet Inter-ORB Protocol: A Brief Description”
http://www.omg.org./library/iiop4.html

[61] Object Management Group, (2002): "OMG IDL Syntax and Semantics" from: "The Common Object Request Broker: Architecture and Specification", v3.0.2, chapter 3, pp. 3-1 - 3-76;
http://www.omg.org/technology/documents/formal/corba_iiop.htm

[62] Object Management Group, (2002): "The Common Object Request Broker: Architecture and Specification", v3.0.2
http://www.omg.org/technology/documents/formal/corba_iiop.htm

[63] OIL specification, http://www.ontoknowledge.org/oil/TR/primitives.html

[64] M. Otte, (1996): "Marketing"; 3. Auflage; WRW Verlag; ISBN: 3-927250-63-5

[65] PABADIS Consortium, (2001): “Agent Fabricator Specification and Conceptual Design”; Task 1.4 Deliverable

[66] PABADIS Consortium, (2001): “Agent Specification and Design”; Task 1.2 Deliverable

[67] PABADIS Consortium, (2001): “Platform Layer and Host Specification and Design”; Task 1.3 Deliverable

[68] PABADIS Consortium, (2001): “The PABADIS Specification - Structure and Behaviour”; Task 6.3 Deliverable

[69] PABADIS Consortium, (2002): “Development of generic agent V1”; Task 2.1 Deliverable

[70] PABADIS Consortium, (2002): “Development of the generic agent V2”; Task 2.2 Deliverable

[71] PABADIS Consortium, (2002): “Development of the machine representation”; Task 2.3 Deliverable

[72] PABADIS Consortium, (2002): “ERP Integration development”; Task 4.1 Deliverable

[73] PABADIS Consortium, (2002): “ERP - Interface and Agent Fabricator development”; Task 4.2 Deliverable

[74] PABADIS Consortium, (2002): “Host Design and Development”; Task 3.3 Deliverable

[75] PABADIS Consortium, (2002): “Platform Testing”; Task 3.2 Deliverable

[76] PABADIS Consortium, (2002): “Plug-and-Participate Capability”; Task 3.4 Deliverable

[77 PABADIS (Plant Automation BAsed on DIstributed Systems) home page www.pabadis.org

[78] Y.K. Penya, S. Deter, T. Sauter, (2003): "Plug-and-Participate Functionality for Agent-Enabled Flexible Automation", IEEE International Conference on Intelligent Engineering Systems (INES), Assiut (Egypt), p. 136.

[79] D. C. Plummer, (1982): "An Ethernet Address Resolution Protocol", RFC 826 http://www.ietf.org/rfc.html

[80] A. Pöschmann, P. Krogel, (2000): "Autoconfiguration Management for Fieldbus - PROFIBUS Plug&Play", Elektrotechnik und Informationstechnik, vol. 117, no.5, 2000, pp. 335-339

[81] PROFIBUS Nutzerorganisation e.V.: "PROFInet - more than just Ethernet!"; http://www.profibus.com/metanavigation/downloads/texte/02029/

[82] PROFIBUS Nutzerorganisation e.V., (2002): "PROFIBUS Technologie und Anwendungen - Systembeschreibung"; http://www.profibus.com/profibus.html

[83] PROFIBUS Nutzerorganisation e.V., (2003): "PROFInet Architecture Description and Specification", v2.0 http://www.profibus.com/

[84] G. Reinhart, (1999): "Wandel in der Montage - Flexibilität ist gut, Reaktionsfähigkeit noch besser" Leitartikel, wt Werkstattechnik 89 (1999) H. 9, pp. 389.

[85] T. Sauter, P. Massotte, (2001): "Enhancement of Distributed Productions Systems through Software Agents"; 8th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Nice, France, pp. 267 - 272; ISBN 0-7803-7241-7

[86] T. Sauter, M. Wollschlaeger, (2000): "Feldbussysteme - Historie, Eigenschaften und Entwicklungstrends", Informationstechnik und Technische Informatik, vol. 42, 2000, pp. 7-16.

[87] G. Seshadri, (2000): "Fundamentals of RMI About this Short Course" http://developer.java.sun.com/developer/onlineTraining/rmi/

[88] SIG Pack Systems, home page; http://www.sigpacksystems.com

[89] L. Smith, C. Roe, K. S. Knudsen, (2002): "A Jini™ Lookup Service for Resource-constrained Devices"; 4th IEEE International Workshop on Networked Appliances

[90] K. Sohr, (2001): "Sicherheitsaspekte von mobilem Code"; ("Security Aspects of mobile Code"); Dissertation, University of Marburg, Department of Mathematics and Computer Science.

[91] M. Sommer, H-P. Gumm, (2002): Einführung in die Informatik, "Objekt Orientiertes Programmieren (OOP)", (Object Oriented Programming (OOP)), pp 207 - 211; 5. Auflage; Oldenbourg Verlag, ISBN 3-486-25050-7

[92] Suda, (1989): “Future Factory System Formulated in Japan”, TECHNO JAPAN, vol. 22

[92] Sun Microsystems: ”Connected Limited DeviceConfiguration - Specification Version 1.1”; http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html

[93] Sun Microsystems: "Java Remote Method Invocation - Distributed Computing For Java" - White Paper; http://java.sun.com/marketing/collateral/javarmi.html

[94] Sun Microsystems: "Java™ RMI-IIOP Documentation"; http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/

[95] Sun Microsystems: “J2ME Building Blocks for Mobile Devices - White Paper on KVM and the Connected, Limited Device Configuration (CLDC)”; http://java.sun.com/products/cldc/wp/KVMwp.pdf

[96] Sun Microsystems: “J2ME CDC Specification v1.0“; http://java.sun.com/products/cdc/

[97] Sun Microsystems, (2001): "Jini™ Technology Surrogate Architecture Specification"; v1.0; http://www.jini.org

[98] Sun Microsystems, (2002): "Java Remote Method Invocation Specification", v1.8; ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf

[99] Sun Microsystems, (2003): "Java™ Object Serialization Specification", sdk 1.4; http://java.sun.com/j2se/1.4/docs/guide/serialization/spec/serial-title.doc.html

[100] The dejay web page: http://www.dejay.org

[101] The LEAP web page: http://leap.crm-paris.com/

[102] The Open Group, (1997): "Technical Standard DCE 1.1: Remote Procedure Call"
http://www.opengroup.org/public/pubs/catalog/c706.htm

[103] TimeSys, (2003): "JTime";
http://www.timesys.com

[104] J. Waldo, (2001): "Mobile Code, Distributed Computing, and Agents"; in: IEEE Intelligent Systems; March/April 2001; pp. 10-12

[105] M. Welsh, (1999): "NinjaRMI: A Free Java RMI - Introduction and Tutorial"
from: UC Berkeley Ninja Project
http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html

[106] L. Widmer, R. Schnider, (2002): “Verteilte Automatisierung mit Jini”, diploma thesis, Hochschule Zurich/Winterthur

[107] S. Willliams, C. Kindel, (1994): "The Component Object Model: A Technical Overview";
Developer Relations Group, Microsoft Corporation
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_comppr.asp

[108] WindRiver, (2002): "JWorks Delivers on the Promises of Java in Embedded Systems"
http://www.windriver.com/products/java/jworks.pdf

[109] WindRiver, (2003): "Unique application environment delivers on the promises of Java in embedded systems"
http://www.windriver.com/products/jworks/jworks4.pdf

[110] G. Wöhe, U. Döring, (2002): "Einführung in die Allgemeine Betriebswirtschaftslehre"; 21. Auflage; Verlag Vahlen, ISBN 3800628651

[111] J. P. Womack, D.T. Jones, D. Roos, (1992): "Die zweite Revolution in der Autoindustrie"; Campus-Verlag, Frankfurt

[112] W3C, (2000): "Extensible Markup Language (XML) 1.0 (Second Edition)";
W3C Recommendation 6 October 2000;
http://www.w3.org/TR/2000/REC-xml-20001006.pdf

Lebenslauf

Name, Vorname	Deter, Steffen
Geburtsort, -datum	Wippra, 07.06.1974
Schulbildung	Polytechnische Oberschule Seebach (Thür.) 1981-1990 Staatliches Gymnasium Ruhla (Thür.) 1990 - 1933
Schulabschluss	Allgemeine Hochschulreife
Hochschule	Diplom Informatik Philipps-Universität Marburg, 12 Semester Oktober 1994 - Juli 2000
Hochschulabschluss	Diplom-Informatiker
Beschäftigungsverhältnisse	August 2000 - September 2000 wissenschaftliche Hilfskraft mit Abschluss Oktober 2000 - November 2003 wissenschaftlicher Mitarbeiter an der Philipps-Universität Marburg

Erklärung

Ich versichere, dass ich meine Dissertation

Plug-and-Participate for Limited Devices in the Field of Industrial Automation

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 in einer ähnlichen Form noch an keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

(Ort, Datum) (Unterschrift mit Vor- und Zuname)