Coordination and P2P Computing

A Thesis Submitted to the College of
Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon, Canada

By
Lichun Ji

Keywords: coordination, distributed computing, Manifold, Peer-to-Peer

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Professor Ralph Deters, for his constant support. Without his help, this work would not be possible. I would also like to thank the members of my committee who attended my defense: Professor JulitaVassileva, Professor Dwight Makaroff and Professor Chris Zhang. Their advice and patience is appreciated. Special thanks go to Professor Dwight Makaroff for his invaluable advice on system evaluation and simulation. I would also like to thank all graduate students of MADMUC labs for their helps.

I dedicate this thesis to my parents.

ABSTRACT

Peer-to-Peer (P2P) refers to a class of systems and/or applications that use distributed resources in a decentralized and autonomous manner to achieve a goal. A number of successful applications, like BitTorrent (for file and content sharing) and SETI@Home (for distributed computing) have demonstrated the feasibility of this approach.

As a new form of distributed computing, P2P computing has the same coordination problems as other forms of distributed computing. Coordination has been considered an important issue in distributed computing for a long time and many coordination models and languages have been developed.

This research focuses on how to solve coordination problems in P2P computing. In particular, it is to provide a seamless P2P computing environment so that the migration of computation components is transparent. This research extends Manifold, an event-driven coordination model, to meet P2P computing requirements and integrates the P2P-Manifold model into an existing platform. The integration hides the complexity of the coordination model and makes the model easy to use.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FORMULAS

LIST OF FIGURES

LIST OF ABBREVIATIONS & TERMS

Condor	A general-purposed distributed computing platform
CORBA	Common Object Request Broker Architecture
C/S	Client/Server Distributed Computing Model
Darwin	A Configuration Language
ECM	Encapsulation Coordination Model (a coordination model)
IWIM	Ideal Worker Ideal Manager (an event-driven coordination model)
JXTA	A P2P Platform
Linda	A Date-driven Coordination Language
MADMUC	Multi-Agent Distributed Mobile and Ubiquitous Computing Laboratory
Manifold	An event-driven Coordination Language
P2P	Peer-to-Peer
SETI@HOME	A P2P Distributed Computing Application. SETI (the Search for Extraterrestrial Intelligence)
SOAP	Simple Object Access Protocol, an XML-based protocol for the exchange of information
STL	Simple Thread Language (hybrid coordination language)
UDDI	Universal Description, Discovery and Integration, service location and contracts advertisement approach
WSDL	Web Services Description Language, an XML-based grammar for describing network services
XML	Extensible Makeup Language
.NET	Microsoft Framework
.NET Remoting	A component of .NET Framework dealing with cross-application domain communication

CHAPTER 1

INTRODUCTION

1.1 Motivation

The continuing growth in processing power, memory and network bandwidth has dramatically changed desktop computing. With the pervasive deployment of desktop machines and increasingly powerful handheld devices, an ever-growing pool of resources is emerging. Currently there are over 400 million computers worldwide of which the majority is often either idle or underutilized.

Recently, the term "P2P" has been used to refer to a collection of applications that harvest the unused processing cycles of desktop computers in a network [1]. A number of successful approaches, like BitTorrent [2], Gnutella [3], and Freenet [4] for file and content sharing and SETI@Home [5] for distributed computing, have demonstrated the feasibility of this approach. Peers (nodes) in a P2P system have equivalent capabilities in providing other parties with data and/or services and cooperate in a decentralized manner.

As a new type of distributed computing, P2P computing has the same coordination problems as other forms of distributed computing, for example: communication, security and synchronization. In the area of distributed computing, coordination in a large community of cooperative heterogeneous components has been a major concern for a long time. This led to the design and implementation of a number of coordination models and their associated languages. These models and languages provide frameworks, which enhance modularity, component reusability, portability and language interoperability [6].

Due to the autonomy and dynamic nature of participants in a P2P environment, it is hard to predict or infer the location and lifetime of the system's resources. In other words, computation components may often need to migrate from location (peer) to

location during execution. This requires additional coordination-oriented features in P2P computing such as autonomous reconfiguration, platform-independent communication and decentralized coordination management.

1.2 Research Statements

This research focuses on how to apply/adapt a coordination model for solving coordination problems in P2P computing. It extends Manifold, an event-driven coordination model, to meet P2P computing requirements and integrates the proposed model into an existing platform. The new P2P coordination model has the following features:

- **Transparency**

  The model needs to hide the migration of components and provide a transparent development and execution environment so that P2P computing application can be treated and developed as normal distributed computing application. This will improve the programmability and adaptability of P2P application.

- **Flexibility**

  The new coordination model should be easy to use and integrate. P2P computing applications are heterogeneous in terms of their programming language, underlying middleware (i.e. software that connects two otherwise separate applications or separate products that serve as the glue between two applications) and running environment. The heterogeneity requires a flexible and adaptable model.

- **Usability**

  The model should add little new work and concepts for programmers developing applications.

P2P computing is viewed here as web-services-style P2P computing. From this perspective, each peer in the network contributes its idle resources through its web services interface. The web services are provided by a (set of) service object(s) and can be either context-independent or context-dependent in terms of the type of the object. Context-independent services are services in which each invocation is isolated. Context-

dependent services are services that contain states and the state of the service object is changed during invocation.

Several assumptions are made in this research:

- The resources of the P2P network are plentiful and there are always enough suitable peers available.
- The selection of participants and the deployment of application components are done prior to the execution and the component migration. In addition, any associated external data components, such as database and data files, are deployed or moved with the component.
- The mobility of coordinators in the model is low. A coordinator is always located on a relatively long-lived host. This assumption is due to the high cost of coordinator migration and the special role a coordinator plays.
- The sudden departure of a participating peer is ignored in this research. A component in the model has enough time for sending necessary messages (e.g. state change messages) and completing a migration. For example, a service provider won't leave until completing any ongoing service call and the migration request of the replacement.
- All participants (hosts) are fully trusted (no security concerns).

1.3 Research Questions

This thesis will focus on the following two main issues:

- **Design & Implementation**
  How to design and implement a coordination model for P2P systems.
- **Evaluation**
  How does the model influence the application and the host machine?

The rest of this thesis is organized as follows: Chapter 2 provides an overview of P2P networks and existing P2P computing systems; Chapter 3 reviews the concepts and works of coordination; Chapter 4 presents the design and the .NET implementation of P2P-Manifold model; Chapter 5 provides the experimental setting and methodology;

Chapter 6 presents the experiments with data analysis; and the thesis finishes with a summary and a discussion on future work.

CHAPTER 2

P2P COMPUTING

"Peer-to-Peer (P2P) refers to a class of systems and/or applications that use distributed resources in a decentralized and autonomous manner to achieve a goal e.g. perform a computation" [1]. The members (called peers) of a P2P network are always in total control of their local resources and can therefore choose to impose or change policies regarding their use. Rather than having *static* and *predefined* roles for the participants like in the client-server model, P2P networks rely on *emerging* and *dynamic* roles as a result of an ongoing self-organization.

The functionalities and application domains of P2P networks lead to four main P2P categories [1]:

- **File sharing**

  File sharing seems to be the most successful application for P2P networks. The basic idea of file sharing is to use the idle disk space for storage and the available network bandwidth for search and download. BitTorrent [2], Gnutella [3], Freenet [4] and FastTrack [7] are just a few of this fastest growing segment of P2P technology.

- **Collaboration systems**

  Collaboration systems allow application-level collaboration among users. These include real-time exchange of message (Project Jabber [8]) and online game/gambling (Zoogi [9]).

- **P2P platforms**

  P2P platforms like JXTA [10] support the developers of P2P applications by offering a wide range of libraries and services (e.g. request routing, peer discovery and peer communication).

- **Distributed computing**

  Distributed computing applications harvest unused processing cycles of computers in the network to delegate and migrate tasks. The SETI@Home project [5], which uses the idle resources of participating peers for its search of extraterrestrial intelligence, is an example of a successful distributed computing application.

2.1 P2P Characteristics

A P2P network is characterized by decentralization, dynamism and heterogeneity. These factors impact the performance and deployment of P2P systems.

2.1.1 Decentralization

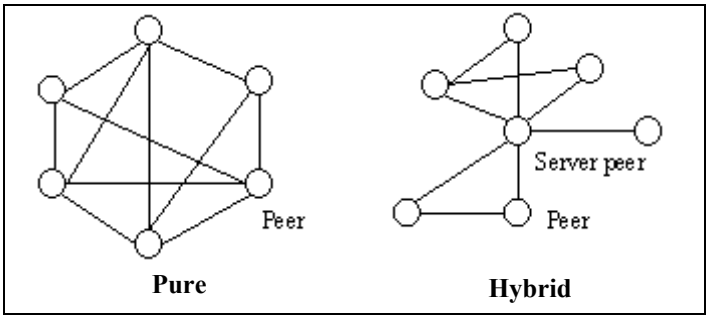


Figure 2-1 High-Level Views of P2P Network

P2P computing provides an alternative to the centralized client/server (C/S) model of computing. A P2P model can be either *pure* or *hybrid* (Figure 2-1). In pure P2P models, such as Gnutella [3] and Freenet [4], all participants (peers) play the same role of both client (service consumer) and server (service provider). In hybrid models, such as BitTorrent [2] and Groove [11], peers first approach a server in order to obtain meta-information, for example: the identity of the destination peer, which offers a required service (data, computation, etc). After this, the P2P communication is performed. The direct service exchange liberates peers from the traditional dependence on central servers. The self-organized peers have a higher degree of autonomy and control over the services they utilize [12].

The decentralization provides the opportunity to make use of unused bandwidth, storage and processing power at the edge of the network. It reduces the cost of system

ownership and maintenance and also improves the scalability. The impact of a peer's entering/leaving will be limited to the directly connected peers instead of damaging the whole network. The P2P model distributes the workload of a server in C/S model and eliminates the single-failure bottleneck of the centralized C/S model.

2.1.2 Dynamism

In a C/S system, the participating components (servers and clients) are predefined and relatively stable during the service period. The P2P computing environment is dynamic, resources, such as compute nodes, will be joining and leaving the system frequently [1]. When an application is intended to support a highly dynamic environment, the P2P approach is a natural fit. For example the Instant Messaging ICQ [13] uses so-called "buddy lists" to inform users when chat friends become available or unavailable. Without this support, a "poll" of chat partners is needed to send periodic status change message [1]. However, the dynamism reduces the Quality-of-Service and increases the complexity, such as dynamic mapping, migration and synchronization.

2.1.3 Heterogeneity

All participating peers are heterogeneous in terms of their compute and storage capacity, and how well they are connected to the other peers. The availability varies widely as some hosts appear and disappear from the network on a regular basis, while others are almost continuously connected. A P2P system has to leverage the heterogeneity to improve robustness and performance.

2.2 Case Studies

Utilizing idle resources is not a new idea. Over time, many different approaches have been developed. In this section, five approaches will be discussed: the *Condor* system, which is one of the earliest systems to harvest the unused resources; the *Avaki* system as an example of the currently emerging grid-oriented approaches; the SETI@Home application; the new Java P2P platform JXTA; and the language-independent .NET platform.

2.2.1 Condor

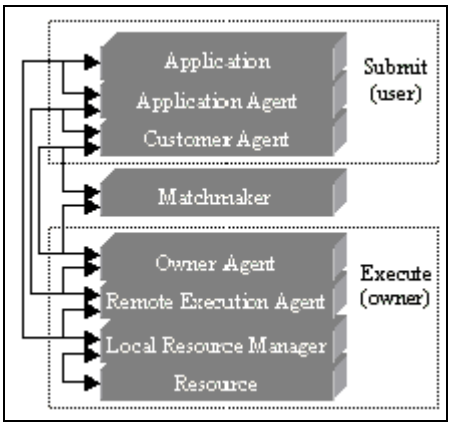


Figure 2-2 The Layers of Condor [14]

Work on Condor started in 1988 at the Computer Sciences Department, at the University of Wisconsin-Madison [14]. Condor aims to offer a general-purpose framework that would allow the use of idle CPU cycles for research purposes. The platform-independent framework provides a basic job queuing mechanism, scheduling policies, priority schemas and resource monitoring and management for distributing "jobs" (tasks) over a pool of machines (peers). It is built on the principle of distributing batch jobs around a loosely coupled cluster of computers to enable a High Throughput Computing (HTC) system. The workflow of a Condor application is as follows (Figure 2-2):

1. Users submit their sets of serial or parallel tasks to Condor in the form of jobs.
2. The Condor matchmaker places jobs into a queue and chooses when and where to run them based on job needs, machine capabilities and usage policies.
3. Condor monitors the progress of jobs and informs the user upon the completion of their jobs.

Condor uses various concepts to ensure fast and safe execution of jobs. To protect the host, all jobs are executed in a restrictive sandbox that prevents/intercepts invoking any system calls. Only "remote" system calls are permitted since they will be executed on the host of the job's owner. In addition to this, Condor supports strong

authentication, encryption, integrity assurance and authorization. To ensure the fast execution of jobs, Condor uses the following techniques [14]:

- **Classified Ads (ClassAd)**

  Ads are used for job/machine mapping, which ensures that the requirements of the jobs fit the capabilities and policy of the machine. A centralized matchmaker performs this mapping. All machines in a Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, the conditions under which it will agree to execute a Condor job and the preferred type of job.

- **Queuing mechanism with priority settings**

  Each user has a Condor queue for all the jobs he/she submitted. The job priority is a means for users to identify the relative importance of individual jobs within a submitted set of jobs. Condor also uses a user priority ranking to determine the amount of pool resources given to the jobs. The higher the priority of the user, the more resources are assigned to his/her jobs.

- **"Flocking" technique**

  Condor supports the linking of independent Condor resource pools. In a linked environment, a Condor pool may transfer a submitted job to another pool that accepts "foreign" jobs.

- **"Up-down" algorithm for scheduling**

  The longer a process runs, the lower its priority becomes. This policy is meant to ensure that users avoid long-lived jobs and the job queues are kept short.

- **Checkpointing**

  Checkpointing is used to compensate for unexpected failures of a host or a job. Condor requires that each job is capable of saving its state in certain time intervals in the form of an image and offers a library to implement this functionality. A checkpoint image contains the process's data and stack segments, as well as information about open files, pending signals, and CPU states. When the job is restarted, the state contained in the checkpoint file is restored. The process resumes the computation at the point where the checkpoint was generated.

According to Condor's usage statistics [15], on a typical day Condor delivers more than 650 CPU days (1 CPU day = 1 CPU×24 hours) to the researchers at the University of Wisconsin-Madison.

2.2.2 Avaki

<table>
<tr><td>Application Services</td><td colspan="2">Job Scheduling, Distributed</td><td colspan="2">Distributed File System</td></tr>
<tr><td rowspan="2">System Management Services</td><td colspan="2">Monitoring, Load Balancing</td><td colspan="2">Policy management</td></tr>
<tr><td colspan="2">Metering, Accounting</td><td colspan="2">Failover + Recovery</td></tr>
<tr><td rowspan="3">Grid Protocol</td><td colspan="4">Identity, Authentication, Encryption, Access Control</td></tr>
<tr><td colspan="4">Scalable Naming and Binding</td></tr>
<tr><td colspan="4">Communication Protocol Adapter</td></tr>
<tr><td>Protocol Adapter</td><td>TCP/IP</td><td>RPC</td><td>JXTA</td><td>.NET</td></tr>
</table>

Figure 2-3 The Layers of Avaki [17]

Andrew Grimshaw at the University of Virginia initiated the Avaki project [16] in 1993, and re-launched it as Avaki Corporation in 2001. Avaki is a grid middleware that enables sharing of data, applications and computing resources targeting the enterprise-wide computing area.

The Avaki grid environment can consist of desktops, workstations, servers and clusters. Each machine in the grid is autonomous and consequently the system management is distributed. Avaki is able to interoperate with queuing systems, load management systems, and/or scheduling systems. Avaki is composed of three services layers (Figure 2-3):

- The grid protocol layer, which provides protocol adapters, security, naming and binding.
- The system management services layer, which provides interfaces for implementing and managing distributed solutions.
- The application services layer, which provides high-level services.

Each resource made available to the Avaki grid has a *unique logical identifier*. Avaki manages grid resources and applications via:

- Access controls - A user or application may or may not have access to a specific service or host computer.
- Matching - Avaki matches application requirements and host characteristics.
- Prioritizing - Avaki evaluates the grid and its application based on polices and load conditions.

To ensure the safe and secure execution of the code, Avaki uses the following approaches:

- **Checkpointing**

  Avaki uses checkpointing to minimize the loss of information in the event of a host or network failure. Hosts, jobs and queues automatically back up their current states.

- **Redundancy**

  Avaki networks are designed to allow the use of redundancy as an additional means for coping with failures. Avaki migrates running applications to another host, based on predefined deployment policies and resource requirements.

- **Authentication**

  The Avaki authentication reduces the need for additional software-based security control, substantially reducing the overhead of sharing resources. Avaki's authentication is based on the resource identity and it uses the Public Key Infrastructure (PKI) technique [17]. It allows the local administrator to control the access to their resources. It also includes user access authorization and resource access authorization.

2.2.3 SETI@Home

SETI@Home was envisioned in 1996 by computer scientist David Gedye, along with Craig Kasnoff and astronomer Woody Sullivan [5]. *SETI* (the Search for ExtraTerrestrial Intelligence) is a collection of research projects aimed at discovering alien civilizations using radio telescopes. Since the analysis of the extensive radio

telescope data (about 35 GB per day) requires significant computing resources, a P2P approach for distributed computing was chosen. SETI@Home engages Internet users around the world in the effort of signal analysis.

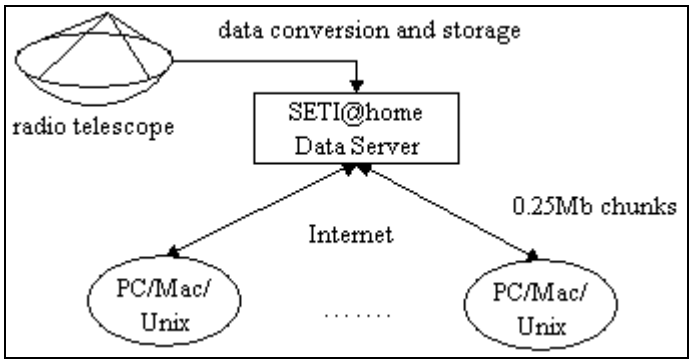


Figure 2-4 SETI@Home Architecture [1]

As shown in Figure 2-4, a central SETI@Home server divides the data into chunks (work-unit) designed for an average desktop computer. Participating peers contact the server and download a chunk of data. After downloading the data, the peer starts processing the data in its idle time (i.e. when the screen-saver is active). The result of the analysis is sent back to the central server and a new cycle of requesting data, processing data and reporting results begins.

The tasks in SETI@Home are independent and can be executed without the need of any network connection. Network connectivity is only needed for receiving data and sending results. The peer data - including the number of work units completed, time of last connection, and team membership - is reported on Web sites allowing users to compete for the biggest CPU contributions. SETI@Home uses a check-pointing mechanism to recover from faults. It saves the dataset and the progress in analyzing it to the hard drive every 10 minutes. To ensure that the hardware and software is working properly, SETI@Home also injects "test signals" into the system. "Suspicious" responses to a work unit or the lack of reported results is recorded and used in evaluating the level of trust assigned to the peer, e.g. preventing the peer from future participation.

The major contribution of SETI@Home is the demonstration of how to apply distributed computing challenges in a P2P network. SETI@Home has managed to

attract several hundred thousand active participants, which hope to be the “one” to discover extraterrestrial. Due to the large number of freely available computing resources, no efforts for optimizing the execution of tasks are necessary.

2.2.4 JXTA

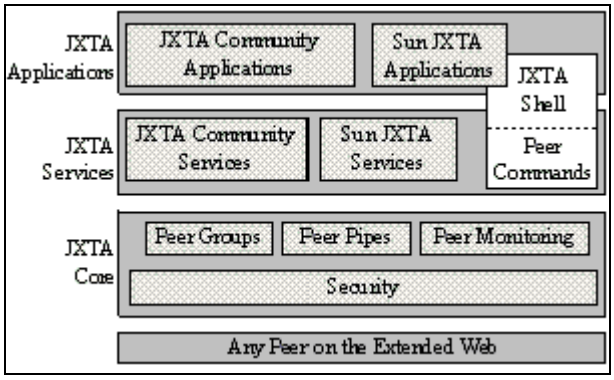


Figure 2-5 JXTA Architecture [18]

JXTA [25] was started at Sun Microsystems in 2001. It is an open-source project (www.jxta.org) and was initiated by Bill Joy to standardize a set of protocols for building P2P applications. JXTA aims at providing a general framework that is software and hardware platform independent. It defines six protocols: Endpoint Routing (ERP), Rendezvous Protocol (RVP), Peer Revolver Protocol (PRP), Peer Discovery Protocol (PDP), Peer Information Protocol (PIP), and Pipe Binding Protocol (PBP). Currently Java and C implementations of the JXTA protocols are available and a .Net version of JXTA is under development.

JXTA has several platform-independent features that make it useful for current P2P application designers (e.g. the Anthill project [19]):

- **Unique IDs for entities and advertisements**

  Each entity (peer, peer group, pipes, advertisement, etc) is assigned and identified by a unique ID. Similar to Condor, all resources in the JXTA network are represented by *advertisements* but the ads in JXTA are XML formatted, making them platform independent and extendable. Peers cache, publish and exchange ads to discover and find available resources. The advertisement mechanism makes all available network resources visible to peers.

- **Concept of peer groups**

  Peers in the JXTA network are linked to at least one *peer group*, which is a dynamic set of peers that share interests and have agreed upon a common set of policies and services. Each *peer group* is a virtual network space consisting of a subset of all devices accessible via an overlay network. The JXTA overlay network is a middleware messaging system designed to allow for end-to-end connectivity between devices across sub-networks.

- **Transparent communication via pipes**

  JXTA uses asynchronous communications channels, called *pipes*, for sending and receiving messages. It offers two modes of communication: point-to-point and propagation. Pipes allow for a simple and transparent form of communication.

- **Rendezvous peers**

  JXTA provides a resolver service based on *rendezvous peers*. *Rendezvous peers* are well-known peers that have agreed to cache a large number of advertisements for exchanging and trading information.

- **Peer-monitoring**

  Peer-monitoring is a core mechanism of JXTA. It enables control of the behavior and activity of peers in a peer group and can be used to implement task management functions for fault detection and recovery.

- **Entry-level trust model**

  Project JXTA provides an entry-level trust model, Poblano [20], which permits peers to either have their own certificate authorities or rely on others.

JXTA provides a general-purpose P2P network programming and computing infrastructure and consequently it supports basic security and communication features. However the support for different computing models is left to the developer of the application.

2.2.5 .NET

Microsoft's .NET (DotNet) [22] was officially launched in June 2000. It provides a platform for developing web services – both P2P and client/server in nature. From .NET's point of view, every computer on the Internet is capable of delivery its own web services.

The design of .NET is focused around decentralization of distributed services. It consists of:

- **.NET framework and .NET compact framework**

  Cross-platform frameworks of classes designed for building and running applications and web services - components that facilitate integration by sharing data and functionality over a network through standard, platform-independent protocols.

- **Developer tools**

  Developer tools provide an integrated development environment (IDE) for maximizing developer productivity with the .NET Framework.

- **Servers & client software**

  They help developers to integrate, run, operate, and manage web services and web-based applications.

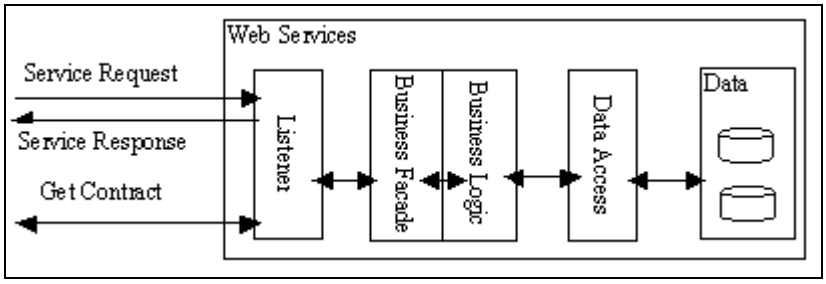


Figure 2-6 Generic Web Service Architecture [21]

The concept of web services is built around web standards and standard protocols, such as *HTTP* [23], *XML* [24], *SOAP* [25], *WSDL* [26] and *UDDI* [27]. The web service architecture is divided into four logical layers (Figure 2-6):

- **Data layer**

  It stores information required by the web service.

- **Data access layer**

  It presents a logical view of the physical data to the business layer. It isolates the business logic from changes to the underlying data stores and ensures the integrity of the data.

- **Business layer**

  It implements the business logic of the web service and is often subdivided into two parts: the *business facade* and the *business logic*. The business facade provides a simple interface, which maps directly to operations exposed by the web service. The business facade uses services provided by the business logic layer.

- **Listener layer**

  It is responsible for receiving incoming messages containing requests for service, parsing the messages, and dispatching the request to the appropriate method on the business facade. If the service returns a response, the listener is also responsible for packaging the response from the business facade into a message and sending it back to the client. The listener also handles requests for contracts and other documents about the web service.

The .NET web services model is implemented in the .NET Remoting Framework (Appendix A). The .NET Remoting Framework provides an extensible framework for objects existing in different application domains (processes) and in different machines to communicate with each other seamlessly. It also offers a programming model and runtime support for making these interactions transparent.

2.2.6 Summary

The previously mentioned systems show a great diversity of approaches. Table 2-1 summarizes the comparison of case studies in system characteristics:

- Condor is a cluster project with a central manager where participating host are known and stable. Each Condor task is independent - no concurrent communication is required.

- Avaki is one of the grid computing platforms. It handles some degree of heterogeneity and dynamism.
- SETI@Home is an application designed for computing independent tasks. It relies on an always-on central server to assign tasks to available voluntary hosts and harvest the results. Therefore the SETI@Home network is heterogeneous and semi-dynamic.
- JXTA standardizes a set of protocols for building P2P applications. All of the protocols concentrate on the network topology and message (advertisement) format. But it leaves the implementation details to the developer. It doesn't mention coordination-oriented issues and the application developers need to design and handle these issues.
- The .NET Framework is a general purpose distributed computing framework. It provides APIs for developing application and treats the coordination supports as a part of application.

Table 2-1 Comparisons of Case Studies

Case	System Characteristics			
	Model	Decentralization	Dynamism	Heterogeneity
Condor	Academic & Open-Source	Cluster with central server	Predictable participant	Low
Avaki	Product & Open-Source	Grid	High	High
SETI@Home	Academic	Central server	Moderate	High
JXTA	Proprietary extensions	Rendezvous peer Peer group	High	High
.NET	Proprietary	N/A	N/A	N/A

As an extension of traditional distributing computing, all of the above P2P computing cases miss one essential issue, coordination. Condor and SETI@Home deal with independent tasks avoiding coordination. Avaki may have considered some coordination issues, though few documents about the design or detailed inner architecture have been released. JXTA and .NET are general-purpose frameworks with protocols and generic APIs. They leave coordination as one of the implementation detail to the developer. Since coordination is essential when cooperating and interacting

among concurrent and distributed components, it is important for a P2P middleware to provide coordination support to the developers.

CHAPTER 3

COORDINATION MODELS AND LANGUAGES

Coordination is a central issue in the design of distributed and parallel systems, which consist of several concurrent, cooperating processes. Carriero and Gelernter first stated the term *Coordination* by advocating the following slogan [28]:

programming = computation + coordination

Programming a distributed or parallel system can be seen as the combination of two distinct activities: the actual *computing* part consisting of a number of processes involved in manipulating data, and a *coordination* part responsible for the communication and cooperation between the processes. Thus, the separation of coordination (communication) and computational concerns allows the separate development and the eventual amalgamation of these two phases.

A *coordination model* is an abstract framework for the composition and interaction of active entities [29]. It encompasses concepts and methodologies for dynamic process creation and destruction, communication mechanism, multiple communication flows and activity spaces. A general coordination model contains the following elements [30]:

- **Coordinated entities**

  Coordinated entities are the active entities (processes) running concurrently. They are the direct subjects of coordination and therefore the building blocks of the coordination architecture.

- **Coordinating media**

  The medium allows the coordination of all participating entities and serves to assemble entities into a configuration. In a shared dataspace model, it is the actual space where coordination takes place.

- **Coordination rules**

  Rules are the specifications of the semantics framework of the model and the definitions of how the entities are coordinated using the coordinating media.

A coordination language is the materialization or the "*linguistic embodiment of a coordination model"* [6]. It offers facilities for controlling synchronization, communication, creation and termination of computational components [28]. A coordination language is not a general purpose programming language. It is orthogonal to a computation language in the sense that it is only concerned with handling the interactions among concurrent activities. Based on a model, a coordination language provides means to compose and control software architectures consisting of concurrent components [30].

Table 3-1 Taxonomy of Coordination Models and Languages [29]

COORDINATION MODELS AND LANGUAGES		
Data-driven ◄----------		----------► Process-oriented
- Linda, Bonita	- Law-Governed Linda	- IWIM
- Laura	- ACLT, TUCSON, MARS	- Manifold, ConCoord
- Objective Linda, JavaSpaces	- IWIM-Linda	- Darwin
	- ECM, STL, STL++, Agent&Co	- Toolbus
- CHAM, Gamma, IAM, Bauhaus		- Contextual Coordination
		- CoLa

Several dimensions can be used to classify the numerous coordination models and languages, for example: the kind of entities that are being coordinated, the underlying architectures assumed by the models, the semantics a model adheres to, etc. The most common classification distinguishes between *data-driven* and *control-driven (process-oriented)* coordination models [6]. There are also hybrid models [29], which merge the two families, by explicitly confronting and integrating elements from one family in the other. Table 3-1 summarizes the taxonomy of some models and languages.

3.1 Data-driven Models and Languages

Coordination models belonging to this category are based on the principle of a shared dataspace (a.k.a. tuple space). A *shared dataspace* [31] refers to the general class of models and languages in which the principal means of communication is a common, content-addressable data structure. Usually there is a (set of) global dataspace(s) shared among components, where components communicate with each other indirectly by posting, reading or withdrawing information to/from it.

Linda, developed in the middle 80's, is one of the first genuine coordination languages in history [6]. It introduced a new paradigm: *generative coordination.* If two processes want to exchange data, the provider generates a new *tuple* (message unit of data without address information) for that data and inserts it into the tuple space from which the consumer can retrieve the tuple. The communication among processes is anonymous - no provider knows the identity of a receiver. Besides passive data tuples, there are active tuples that represent processes. Linda provides a set of simple coordination primitives:

- out(t) - puts a new passive tuple in the dataspace;
- eval(t) - puts an active tuple (process) in the dataspace;
- in(t) - retrieves a passive tuple from the dataspace;
- rd(t) – creates and retrieves a copy of a passive tuple in the dataspace;
- inp(t), rdp(t) - non-blocking variants of in(t) and rd(t).

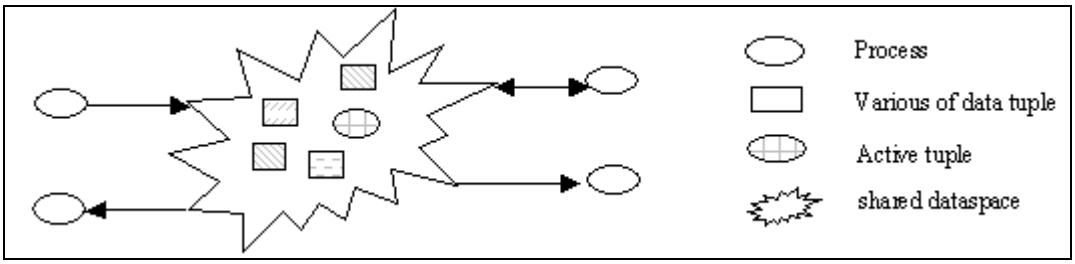


Figure 3-1 Shared Dataspace Principle [31]

Linda has inspired the creation of many other similar languages, such as Bonita, Objective Linda, GAMMA, etc [6]. *JavaSpaces* [32] is a Java-based Linda-like model introduced by Sun Microsystems and used in the JINI framework [33].

The concept of data-driven coordination is simple to understand and implement. But data-driven models have two significant disadvantages. One is the network overhead due to tuple searching and data duplication. Therefore it is not suitable for large systems with many active processes. Another problem is the lack of a clear separation between the coordination functionality and the purely computation functionality of a process. This means that programmers may have to mix coordination and computation code within the process definition, which decreases the components' reusability and portability.

3.2 Process-oriented Models and Languages

Within process-oriented (or event-driven, control-driven) coordination models, "the state of the computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to" [30]. Processes send out control messages or events to their environment (i.e. other interested processes). These messages/events are used to notify others of their state or to inform them of any state changes. The main goal of the process-oriented model is to separate computation and coordination so as to increase reusability. The components being coordinated are considered as black boxes that produce and consume data via well-defined interfaces to the external world. Data is transported by connections between provider and consumer components.

3.2.1 Manifold

Manifold [34] is an event-driven coordination language based on *state transitions.* The concept model behind Manifold is the IWIM (Ideal Worker Ideal Manager) model.

IWIM model

The IWIM (Ideal Worker Ideal Manager) communication model completely separates the computational aspects of a process from its communication aspects, thus

encourages a weak coupling between worker processes in the coordination environment. The basic concepts in the IWIM model are *processes*, *events, ports* and *channels*.

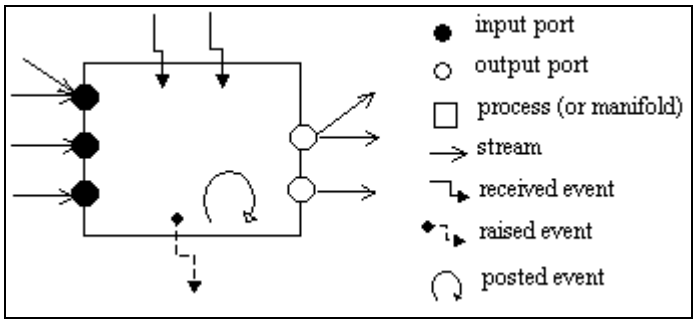


Figure 3-2 IWIM Process Model [34]

A *process* is a black box with well-defined *ports* of connection through which it exchanges *units* of information with its environment. There are two different types of processes: *managers* (or *coordinators*) and *workers*. A *manager* is responsible for setting up and managing the communication needs of a group of workers it controls. A *worker* is unaware of other processes in the communication channels (i.e. where data is sent to/received from). Worker processes can themselves be managers of subgroups of other processes. More than one manager can coordinate a worker's activities as a member of different subgroups.

Events are raised by their source and can be consumed by interested processes in the same environment. Events are used for sending process state information to other processes. Events are non-parameterized and are used only for triggering state transitions, causing the evolution of the coordinated apparatus. Every process in an environment may capture events occurrence and react to them.

Ports are named openings of processes through which data is exchanged. Each port is used for the exchange of data in only one direction. *Channels* are directed interconnections between ports.

Manifold language

Manifold is a coordination language based on the IWIM model [18]. Every basic IWIM abstraction represents a Manifold language construct. In particular, a stream in Manifold corresponds to the asynchronous communication channels in the IWIM model.

A *process* (also called a *manifold*) is an independent, autonomous, activity entity that executes a procedure. Processes are unaware of the environment and influenced by other processes through their ports and events. There are two types of manifold, *atomic processes* that are programs written in a programming language other than Manifold, and *regular manifold*s that are written in the Manifold language. Atomic processes and regular manifolds in Manifold correspond to the workers and managers in the IWIM model. A regular manifold consists of several states, which are composed of a label and a body. The label is a condition for the state transition and the body is a set of actions. At any moment, a Manifold process must be in one and only one state. From this perspective, the Manifold coordination topology is event-driven and based on state transitions. A state is *preempted* if the occurrence of an event that matches the label and another state can cause a state transition during its execution.

An *event* is an asynchronous, non-decomposable (atomic) message, broadcasted by a process in its environment or posted internally within the process. The environment of a process includes all running processes in the same application. Events are identified by their names and source processes. An observer process is responsible for consuming interesting events in its environment and deciding how to react to that event. Occurrences of events that are of interest to a process can be *ignored, saved* (remains in the event memory to be handle at a later time) or be used for a state transition.

A *port* is a means by which information produced by the process is exchanged with other processes. A port is uni-directional and can be either *input* or *output*. All process instances in Manifold have three default ports *input, output* and *error*. A *stream* connects one source port and one sink port. But one port can bind any number of streams. Manifold supports KB (Keep-Break), BB (Break-Break), KK (Keep-Keep), and BK (Break-Keep) streams with infinite FIFO (First-In-First-Out) queues. Only KB and BK streams can be re-connected. “Keep” and “Break” attributes represent the connection types of ports. “Keep” means that one end of the stream does not disconnect, even if the stream realizes that the connection at its opposite end is broken. “Break” means that the stream is disconnected once the opposite end of the stream is broken.

```
export manifold Sorter()
{
   event filled, flushed, finished.
   process atomsort is AtomicSorter(filled).
   stream reconnect KB input->*.
   priority filled<finished.
   begin:
   (
      activate(atomsort), input->atomsort,
      guard(input,a_everdisconnected!empty,finished)
   ).
   finished:
   {
      ignore filled.
      begin: atomsort->output
   }.
   filled:
   {
      process merge<a,b|output> is AtomicIntMerger.
      stream KK *->(merge.a,merge.b).
      stream KK merge->output.
      begin:
       (
         activate(merge), input->Sorter->merge.a,
         atomsort->merge.b,
         merge->output
      ).
      end | finished:.
   }.
   end:
   {
      begin:
       (
         guard(output,a_disconnected,flushed),
         terminated(void)
      ).
      flushed: halt.
   }.
}
manifold Main
{
   auto process read is ReadFile("unsorted").
   auto process sort is Sorter.
   auto process print is printunits.
   begin: read->sort->print.
}
```

Figure 3-3 A Manifold Example [28]

Figure 3-3 is a Manifold example of merge sort. *Sorter* activates the actual sorting. *AtomicSorter* and *AtomicIntMerger* are responsible for sorting and merging the output of sorting respectively.

3.2.2 Darwin

Darwin is a configuration language that enables the specification of systems as a collection of components and their interconnections (services) [35] and is based on its predecessor, the Conic configuration language [36]. Darwin encourages a component-based approach to program structuring in which the unit of structure (*component*) hides its behavior (services it provides and requires) behind a well-defined interface. *Composite components* are constructed by combining (in parallel) more elementary components. The general form of a Darwin program is therefore the tree (Figure 3-4) in which the root and all intermediate nodes are *composite components*; the leaves are primitive *components* encapsulating behavioral as opposed to structural aspects [37].

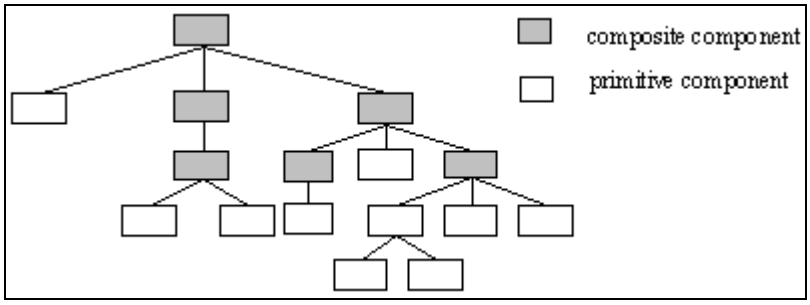


Figure 3-4 Darwin Tree Constructor View [37]

Components and Services

Components in Darwin are black boxes with the interactive interfaces of services they *provide* to other components and services they *require* from other components. The names of services are local to the component type specification, which avoids knowledge of the external world. The context-independent property not only reduces the components' implicit dependencies on their environment but also increases component's re-use and replacement abilities.

Services are identified by their owners (providers), service names and service types (provide or require). Darwin supports service names with wild characters (*), which allows one-to-many, many-to-one and many-to-many interactions (service binding) beyond the basic one-to-one communication.

Composite Components

Composite components (Figure 3-5) are constructed from both computational components and other composite components. The overall system is a hierarchy of structured composite components with a collection of concurrently and distributed (primitive) components. Services that one component provides are visible to the composite component that contains it. Besides the basic component declarations, composite components instantiate components they contain as well as bind the services between these components and themselves. The binding is only made between required and provided services with comparable type.

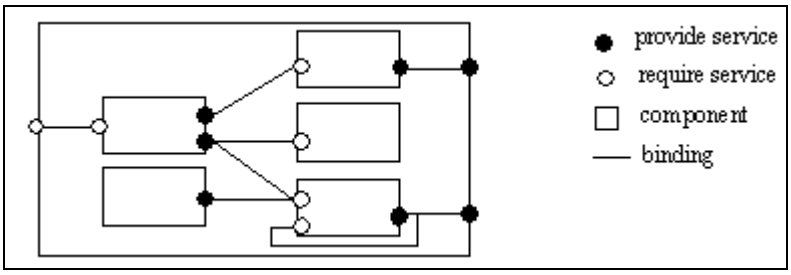


Figure 3-5 Composite Component [37]

3.3 Hybrid Models and Languages

Hybrid models merge functionalities of shared dataspace and process-oriented models by explicitly confronting and integrating elements from one family in the other.

3.3.1 STL

STL (Simple Thread Language) is a hybrid coordination language. It is based on the coordination model ECM (Encapsulation Coordination Model). STL aims to provide a framework for distributed MAS (Multi-Agent System [38]), which is made up of autonomous interactive agents.

ECM Model

ECM is a model to coordinate distributed application components and concentrates only on coordination issues. It integrates shared dataspace functionalities in a process-oriented view [29]. ECM uses an encapsulation mechanism as its primary

abstraction (blops). The mechanism offers structured and separate namespaces that can be hierarchically organized. Within these blops, active entities communicate anonymously through connections. The connections are established by the matching of the entities' communication interfaces (ports) [39].

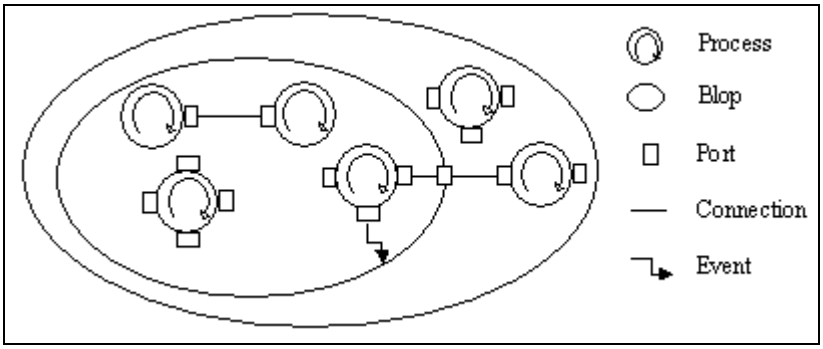


Figure 3-6 The ECM Coordination Model [39]

Figure 3-6 is an overview of the programming metaphor used in ECM. ECM includes five building abstractions:

- **Blops**

  Blops are groups of processes and ports.

- **Processes**

  Processes are black-box style components which perform specific activities but only interact via specified interfaces. They are almost the same as Manifold processes.

- **Ports**

  Ports are endpoints of processes/blops to establish connections to the external world. Ports belong to same level of abstraction but different objects, are matched by their signatures (names and a set of features).

- **Connections**

  Connections are established among matched ports. There are three generic types: Point-to-Point stream, Group and Blackboard.

- **Events**

  Events are attached to conditions on ports. They are triggered by check conditions attached to port states and handled by event handlers inside the blops.

STL Specialties

STL is a realization of the ECM model applied to a multi-threaded application on a LAN of UNIX workstations. The implementation of STL resides on the top of the PT-PVM [40]. Blops are implemented as heavyweight UNIX processes, and processes are implemented as lightweight processes (threads).

A blop is defined as a process with a name and body, in which ports and inner entities are defined. A blop object, which is an instance of a named blop process, can be placed onto any specific physical working unit. Blops can also be nested (Figure 3-6).

STL processes can be activated within the coordination language (through the instantiation of a process object inside a blop) or in the computation language. A process terminates implicitly once it is computed.

STL allows two kinds of ports, *static* ports as interfaces of entities and *dynamic* ports that are defined ahead and created at runtime. A port type is identified by its attributes (communication, orientation, capacity, etc). STL provides only asynchronous communication between ports.

An event's condition check is executed by the system every time data flows through the bound port or a process accessed the bound port. Each installed event handling routine is unloaded by the blop after handling the corresponding event. The handler must be reinstalled for dealing with same event again.

3.4 Comparison and Discussion

3.4.1 Data-driven vs. process-oriented coordination models

The main difference between data-driven and process-oriented models is the degree of separation of computation from coordination. The process-oriented models completely separate computation and coordination modules. This kind of model is only responsible for the coordinated patterns in which they are involved. The actual values of the data being manipulated by the processes are almost never involved. Processes using data-driven model manage both the values of the data being transferred and the actual configuration of the coordinated components. In other words, processes are responsible

for "*both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanism each language provides*" [6].

Another difference is the coordination unit. The data-driven models tend to coordinate data whereas the control-driven models tend to coordinate entities (processes). In the former model, a programmer has more control over the manipulated data. Therefore the data-driven model is suitable for coordination problems in parallel computing and process-oriented model serves primarily for modeling systems [6].

3.4.2 STL, Darwin vs. Manifold

STL, Darwin and Manifold are all used in different application domains. STL is designed for an autonomous agents system. It is an extended process-driven coordination language with shared space concepts, such as blackboard communication. Darwin, a high-level configuration language, focuses more on component instantiation, and Manifold is a purely control-driven language for scientific computing. It fully separates the coordination and computation so as to increase the reusability of code. The summarized comparison of these three models is listed in Table 3-2.

3.4.3 Coordination in P2P computing

All the mentioned models and languages focus on traditional distributed computing domain in which the participating machines are relative stable. Since a P2P environment is dynamic. Heterogeneous and decentralized, it requires additional coordination oriented features:

- **Autonomous reconfiguration**

  The instability of participants causes frequent component migration. The effect of the unexpected migrations to all interactive components should be minimized. The migration should be transparent to the developer.

- **Platform-independent communication**

  The participating peers are heterogeneous, e.g. offer different system resources and security policies.

- **Decentralized coordination management**

  Every peer in the P2P network is volatile. This requires the decentralized management so that to avoid single-point failure.

- **Programmability**

  It should be possible to easily integrate the coordination model into existing applications.

The separation of worker and manager of the process-driven coordination models, especially Manifold, will be more suitable in P2P computing than a data-driven model. It provides the abilities of easy reconfiguration and decentralization the coordination work to several peers. It has therefore been chosen as the basis of a P2P coordination model.

Table 3-2 Comparisons of STL, Darwin and Manifold

Coordination language	STL	Darwin	Manifold
Model	ECM	Conic configuration language	IWIM
Encapsulation	Blop	Composite components	Nested process
Entity	Signature: name and ports Two kinds of entities: process and blop	Signature: Name and services Two kinds of entities: component and composite component	Signature: name, ports and events Two kinds of entities: Manifold and Process
Interface	Port is the interface of process and blop Port identified by name and features (Point-to-point, group & blackboard)	Services provided and required	Port at the boundary of process Port is either input or output Port has name and connectivity type: *, break and keep
Connection (Stream)	Stream connects matched ports Asynchronous communication	Stream connects between provided service and required service	Stream connects between ports Stream can be reconnectable Stream is reliable, directed and buffered flow
Event	Attached to condition on ports Handled inside blop	N/A	Asynchronous and atomic message Raised or posted by process Identified by event name and their source Handled by observer process
Application Domain	Autonomous agents system	Distributed system's architectural configuration	Scientific computing and S/W architecture
Implementation	Multi-threaded base on PT-PVM UNIX	C++ oriented Unix based	C implementation Multi-platform
Decoupling	Separate language	Separate fully fledged coordination component	Separate language

CHAPTER 4

P2P-MANIFOLD

P2P-Manifold is an extension of the Manifold coordination model, designed to meet the requirement of the P2P environment: the transparent and seamless component migration. Manifold was selected due to the following features: the full separation of computation and management and the black-box components mechanism. These features are ideal for developing distributed applications in the dynamic and heterogeneous environment of a P2P network.

P2P-Manifold is developed as an embedded layer for an existing middleware. It provides a transparent environment for P2P programming (i.e. it is programmable without additional knowledge of the underlying coordination model and is easy to migrate existing distributed applications into the P2P environment), and supports seamless component migration so that the instability of a participating peer won't affect the execution.

4.1 Components

The P2P-Manifold model contains two kinds of components, *coordinator* and *computation components*. Each component has a unique name, a unique URL address and provides services. A component's name is kept unchanged once created, while the URL changes after migration. The communications among all components are through web services. The workgroup for an assignment is organized as an m-ary tree (Figure 4-1). One component is managed by one and only one coordinator, while one coordinator can manage more than one component. A component only communicates with its managing coordinator and is unaware of the rest of the environment. When a component

moves, it alone is responsible for sending a migration message to its managing coordinator and managed components (if it is a coordinator).

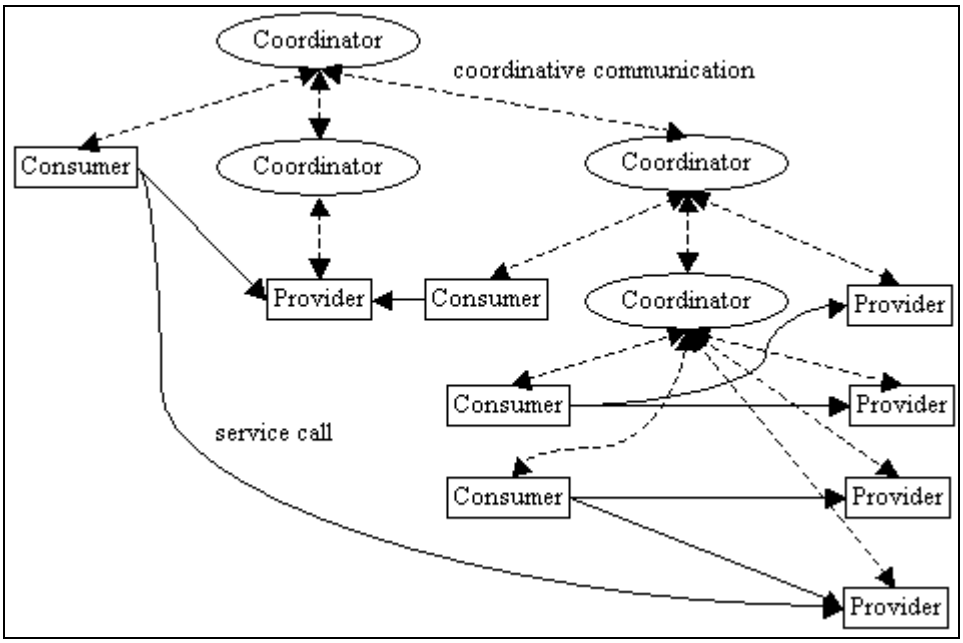


Figure 4-1 P2P-Manifold Architecture

Coordinator

A coordinator works as a redirection manager and a service registry server to its managed components. Coordinators keep track of managed components' unique name, location and status. The status contains two types: suspend and ready, which indicates the communicable status of a responding component/service.

When a managed provider changes, the coordinator is responsible for updating the provider registration record and informing the associated consumers of the change. The associated consumers comprise all real and logical consumers, which called the service before. The consumer/provider interaction relationship is recorded by the coordinator when a consumer asks it to search for a service.

When a consumer migrates (i.e. changes the status to "suspend"), the managing coordinator creates a temporary buffer for a suspended consumer and caches all messages to that consumer. The cached messages are sent to the consumer once it resumes.

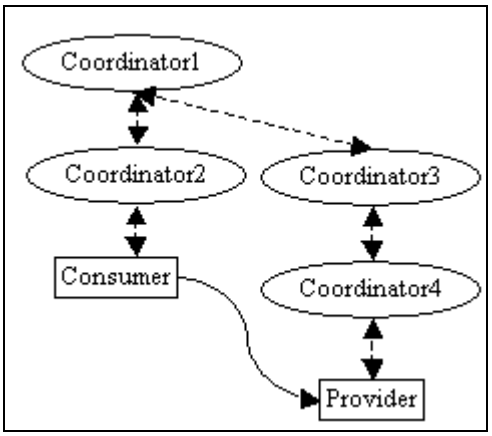


Figure 4-2 Cross-Coordinator Service Example

A coordinator registers the services provided by the managed providers as well as the cross-coordinator services where the current coordinator was in the route between a consumer and a provider. For cross-coordinator service invocation, each midway coordinator is recorded as a logical consumer to the previous coordinator. Figure 4-2 is an example of a cross-coordinator service. Coordinator2 is a logical consumer to Coordinator1 while Coordinator1 is a logical provider to Coordinator2. When the Provider moves, a migration message is sent from the Provider to Coordinator4. Coordinator4 updates its registry table. Since Coordinator3 is recorded as a logical consumer for a service Provider published, it sends Coordinator3 a service update message. Every midway coordinator forwards a service update message until it reaches the Consumer.

Computation Component

Computation components perform the computation assignments in a black-box style. Each computation component registers to one coordinator at startup. In the P2P-Manifold model, computation components are divided into *service providers* and *consumers* in terms of their functionality. Service providers publish services and register the services with the managing coordinator (Figure 4-3). Consumers obtain the location of a service from the coordinators prior to the first invocation and cache the location of called services locally. Further provider changes are pushed automatically from the

managing coordinators to it. The push methodology effectively reduces the network overhead caused by busy checking (i.e. client pulling).

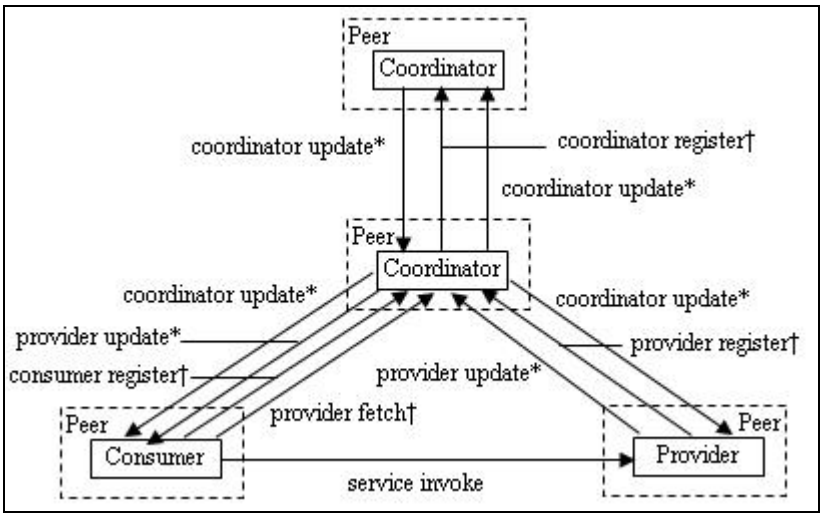


Figure 4-3 P2P-Manifold Interactions

***: Happens only if the provider migrates;**

†: Happens once when new provider comes or first service call made

4.2 Services

In terms of the functionalities, the P2P-Manifold model includes two kinds of services: computation services and coordination services.

Computation Services

Computation services are the web services offered by a service provider. A consumer invokes a computation service in two steps: service searching and service invoking. The search for a service is the search for the provider's resource location. It starts from the consumer's local cache, which contains the location of already invoked services. If no local record is found, the consumer contacts its coordinator to extend the search. The coordinator first searches locally, then forwards the query to all managed coordinators and its managing coordinator if the service is not found locally.

Coordination Services

Each component in the model provides a set of coordination services for coordination communication:

- **Migration services**

  Migration services allow the new replacement to clone state data from a leaving component.

- **Update services**

  Except for the root coordinator, all other components provide an update interface for the managing coordinator. Consumers have additional interfaces for service updates, which are called by the managing coordinator when service provider moves.

A coordinator also provides services for managed components registration when component startup and service for update after component migration. In addition, it provides services to consumer and other coordinators for service search.

4.3 Migration

There are two types of migrations in P2P-Mnaifold model: coordinator migration and computation component migration. The migration occurs when a peer will no longer contribute and a new replacement has been found. Basically, the migration of a component contains three steps:

- Notifying the suspension of a current component. A computation component needs only to notify its coordinator while the migration of a coordinator consists of notifying all connected components. The connected components vary from coordinator to coordinator and might include managing coordinator, managed coordinators and managed computation components.
- Cloning the component onto the new host to maintain the same state for continuity.
- Notifying the resume of the component to all connected components.

4.4 .NET Implementation

A .NET Framework implementation of the P2P-Manifold model was developed as a set of dynamically linked libraries (DLL) using the C# programming language. With .NET's cross-language support, the library can be integrated to all the .NET Framework supported program languages (e.g. C++, VB, Fortran, Java Language, etc [38]). Figure 5-4 is the architecture of this implementation and the information of the completed libraries can be found in Appendix B.

Each remote communication mechanism among components is an XML web service. The open XML web services architecture allows programs written in different languages on different platforms to communicate with each other in a standards-based way.

Coordinator

Except for the root coordinator, each coordinator registers with a managing coordinator. A coordinator contains two local registry tables:

- **Service registry table**

  The service registry table maintains the services provided by the managed providers and any traced cross-coordinator services. The service record contains the service object's name and URL. It is the local index table for later service searching.

- **Interaction registry table**

  The interaction registry table maintains the consumer-service relationships. The information is traced by the coordinator when a (logical) consumer asks it for a service. The table is used for later pushing provider update message to consumers.

Local Proxy

The local proxy is a local routing server for consumer application(s) with a predefined web-services-style interface. It is responsible for communicating with managing coordinators and caching the provider's information in its local registry table. The migration of a local proxy includes the cloning of the registry table and the managing coordinator's information.

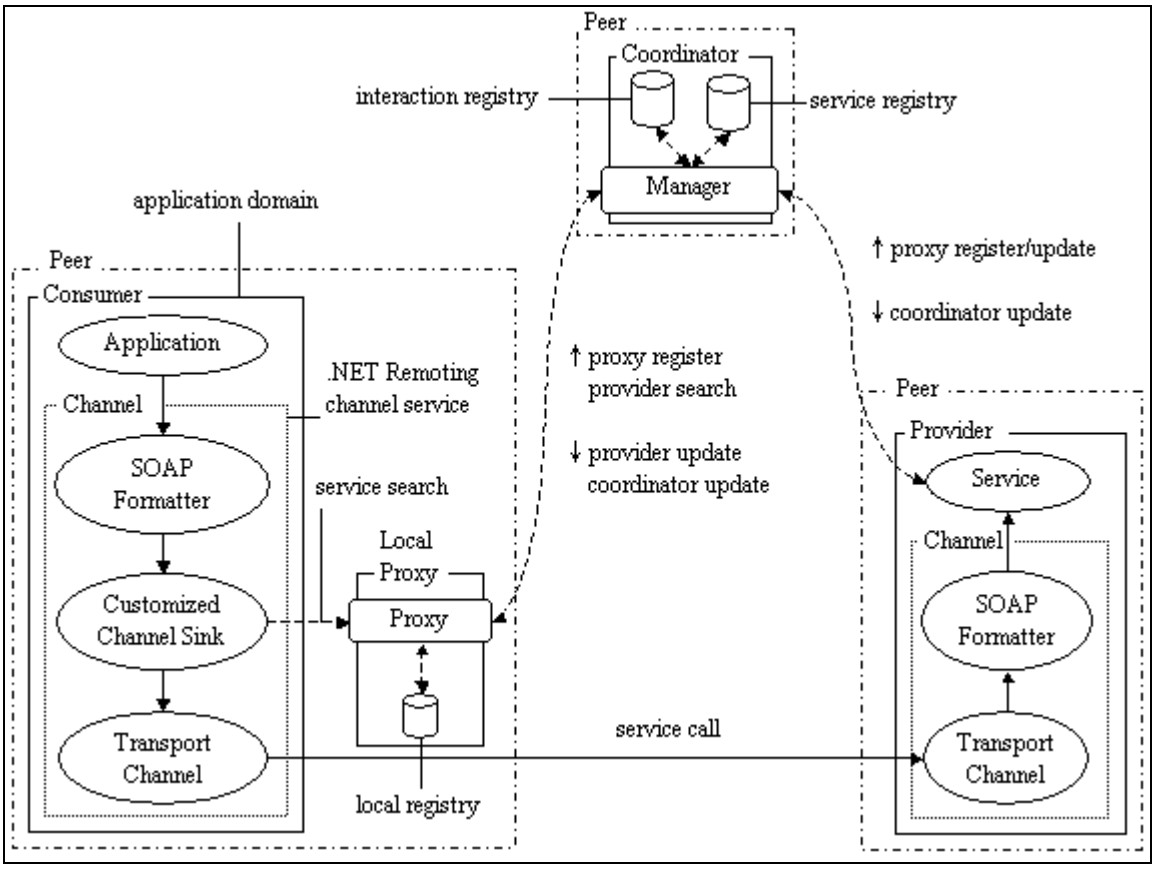


Figure 4-4 Architecture of .NET P2P-Manifold Implementation

Consumer

The consumer is a normal web services application that consumes the computation services. Instead of making the service calls to the real service provider, all calls are made to the local proxy. The local proxy searches the request service and returns the associated provider's location. The redirection of the service call is made by the underlying custom sink bound to the consumer's channel for outgoing service.

Provider

The provider is a normal web service application with additional responsibility for registering/registering all its services to the managing coordinator when startup or after migration. The registration is via a pre-implemented service assistant class. Service objects in a P2P-Manifold workgroup must have unique names since the name is used to identify the service object.

CHAPTER 5

EXPERIMENTAL SETUP AND METHODOLOGY

5.1 Data Collecting

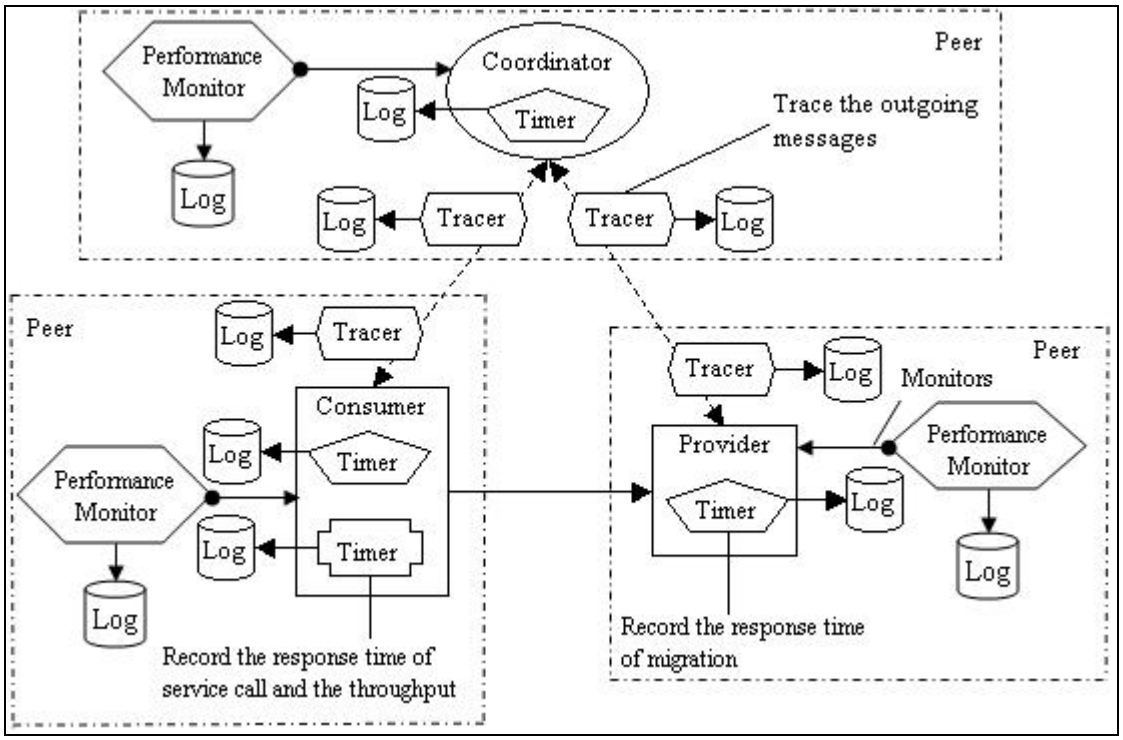


Figure 5-1 Data Collecting Model

The P2P-Manifold model is evaluated by a series of experiments, which collect three types of data (Figure 5-1):

- **The response time of service invocation and component update**

  This kind of experiment is used to measure the latency of service calls and the system throughput. Since all services in the P2P-Manifold model are synchronized, the response time is traced in the original sender side. A throughput counter is deployed on the consumer side to count the total completed calls within a given time unit.

- **The system performance data**

  The application's performance data is used to measure the system overhead caused by the model. The data includes CPU and memory load. An external performance monitor is developed to measure the performance of applications with and without the P2P-Manifold model. The comparison of the above two situations measures the system overhead caused by the model.

- **The size of the coordination messages**

  The message size is used to measure the network overhead. A communication between connected components is composed of a service request and a response. A message tracer is developed and connected to the channels of the coordinator. It intercepts the message and records the message to log file after receiving and before returning.

5.2 Performance Metric

The performance of the P2P-Manifold model is measured by the following metrics:

- **System overhead**

  The overhead here is composed of the system overhead caused by the model and the network overhead of coordination communications. The model should be lightweight with minimized system and network overhead.

- **System latency & throughput**

  The system latency refers to the latency of computational service invocations and the throughput refers to the count of computational services completed within a unit of time. An efficient system should minimize the service latency and increase the throughput.

- **Programmability**

  The programmability is measured by two factors: development difficulty and execution transparency. The model should minimize the efforts to migrate distributed computing applications into the P2P environment.

- **Component Availability**

  The availability here refers to the availability of the computational services provided in the workgroup of an assignment. The more available the service is, the higher the performance of the system will be.

5.3 Theoretical Analysis

This section analyzes the network overhead and the network latency from the perspective of one isolated service invocation.

5.3.1 Network overhead

In the P2P-Manifold model, there are two kinds of service invocations: non-cross-coordinator and cross-coordinator invocation. The network overhead of a non-cross-coordinator service invocation is:

$$O_{service} = (n-1)*O_{local\text{-}fetch} + O_{remote\text{-}fetch} + O_{provider\text{-}register} + 3/2*p*O_{provider\text{-}update}^{\pm} + O_{consumer\text{-}register} + 2*q*O_{consumer\text{-}update} + 2*r*2*O_{coordinator\text{-}update} \quad (1)$$

where:

$O_{local\text{-}fetch}$ is the network overhead caused by the searching of the local routing table ("n" is the number of service calls).

$O_{remote\text{-}fetch}$ is the network overhead caused by a remote (coordinator) service search.

$O_{provider\text{-}register}$ is the network overhead caused by provider registration.

$O_{provider\text{-}update}$ is the network overhead caused by provider migration ("p" is the migration time). The messages are sent from the provider to the coordinator and the coordinator forwards them to the consumer. The average of a context-independent service is p. The average of a context-dependent service is 2*p where one migration operation includes two messages, one for "suspend" message and other for "resume" message.

$O_{consumer\text{-}register}$ is the network overhead caused by consumer registration.

$O_{consumer\text{-}update}$ is the network overhead caused by consumer migration ("q" is the number of migrations). One migration operation includes two messages, one for "suspend" message and another for the "resume" message.

$O_{coordinator\text{-}update}$ is the network overhead caused by coordinator migration ("r" is the times). The migration affects both the provider and the consumer. One migration operation includes two messages, one for "suspend" message and other for "resume" message.

The message size is one of the main factors affecting the message transport speed and the network overhead. Each coordination message uses the same format and the size of message is almost the same. There are two types of communication overheads in the system: local (local provider fetch) or remote (all the others). From formula (1), the overall overhead can be expressed as:

$$O_{service} = (n + 3/2*p + 2*q + 4*r +2)*O_{message} + (3/2*p + 2*q + 4*r +3)*O_{transport} \quad (2)$$

Cross-coordinator service invocation requires additional communication among coordinators. The cross-coordinator service request is forwarded between coordinators until it is found. Each coordinator keeps track of each required cross-coordinator service. Therefore, the worst case of the cross-coordinator search is h (i.e. reaching all coordinators in the workgroup) and the average is (1+h)/2.

Extending formula (1), the average network overhead of both non-cross-coordinator and cross-coordinator service invocation is:

$$O_{service} = (n-1)*O_{local\text{-}fetch} + (1+h)/2*O_{remote\text{-}fetch} + O_{provider\text{-}register} + 3/2*\ p*O_{provider\text{-}update} + O_{consumer\text{-}register} + 2*q*O_{consumer\text{-}update} + 2*[r1 + r2 + \ldots + rm]*2*O_{coordinator\text{-}update} \quad (3)$$

where:

n: the amount of service calls between consumer and provider.

h: the number of coordinators in the workgroup for an assignment.

m: the number of coordinators in the search chain, including direct manager coordinators of consumer and provider.

r1...rm: the migration time of midway coordinators ("r1" represents the migration time of a direct manager coordinator of a consumer and "rm" is for direct manager coordinator of provider).

p: the migration times of provider.

q: the migration times of consumer.

Similar to the simplification from formula (1) to formula (2), formula (3) can be simplified to be:

$$O_{service} = [n + 1 + (1+h)/2 + 3/2*p + 2*q + 4*(r1+\ldots+rm)]*O_{message} + [2 + (1+h)/2 + 3/2*p + 2*q + 4*(r1+\ldots+rm)]*O_{transport} \quad (4)$$

5.3.2 Network Latency

One service call contains two steps: searching the provider's location and making the service call. Providers and consumers need to register their service objects or local proxies when they startup or migrate. Excluding the one-time or random communications, the response time of each service call is:

$$T_{service} = T_{local\text{-}fetch} + \alpha * T_{remote\text{-}fetch} + T_{invocation} \quad (5)$$

where:

$T_{local\text{-}fetch}$ is the time for searching a provider's URL locally.

$T_{remote\text{-}fetch}$ is the time for searching a provider's URL from all coordinators of the workgroup ("α" represents the percentage of participating coordinators in the workgroup for the search). It equals 0 for later service calls where the provider info can be obtained locally.

$T_{invocation}$ is the time of real service invocation.

The response time of each web service is:

$$T_{web\text{-}service} = T_{request} + T_{execution} + T_{response} \quad (6)$$

where:

$T_{request}$ is the response time for a request message encoding/decoding and transport,

$T_{execution}$ is the response time of the service execution in server side, and

$T_{response}$ is the response time of the response message encoding/decoding and transport.

CHAPTER 6

EXPERIMENTS

Three sets of experiments have been conducted to evaluate the model in different aspects:

- **Throughput and response time experiments**
  These experiments evaluate the impact of the P2P-Manifold model on the response time of service calls and the service throughput.
- **System performance experiments**
  Performance experiments monitor the system performance of the local proxy and the coordinator. The performance data includes CPU time and memory load.
- **Coordination message experiments**
  In this set of experiments, the coordination messages are analyzed to calculate the network overhead caused by the model.

6.1 Throughput and Response Time Experiments

System throughput and latency are important when measuring the system efficiency. These experiments measure the throughput and service latency of applications developed with the P2P-Manifold model. The throughput and latency are affected by three factors: the status of components (i.e. stable or mobile), the organization of workgroups and the number of participating components. Therefore, several different test situations are designed to measure the impact of these three factors.

The data is collected by the built-in monitor of the consumer application and recorded in log files. For each experiment, two kinds of context-independent services are tested: simple services and complex services (in terms of the completion time of service). Each mobile component migrates in a predefined time interval, while coordinators migrate less frequently than the computation components.

The throughput is measured over a period of five minutes. Due to the delay in simultaneously starting all components, components may start with up to five seconds delay. Therefore, there may be a 1.7% error when counting the throughput of a five-minute period. In all tests, the consumer continuously calls services provided by the provider. Each experiment has run twice on pool machines in MADMUC lab [41].

Since the two tests were run on same machines and the results of them are similar, only one set of test data is presented. The first 50 service calls of each test are used to analyze the response time. For better representation, the Y-axis data points in all the figures of the section that presenting the service response time (Figure 6-2, 6-4, 6-7, 6-9 and 6-12) has been calculated using the following formulas:

- Simple Service: $y = \log_{10} (y - 1000)$
- Complex Service: $y = \log_{10} (y - 5000)$

A consequence of this scale is that the separation between the smaller values is greatly exaggerated, while the larger values appear much closer on the graph.

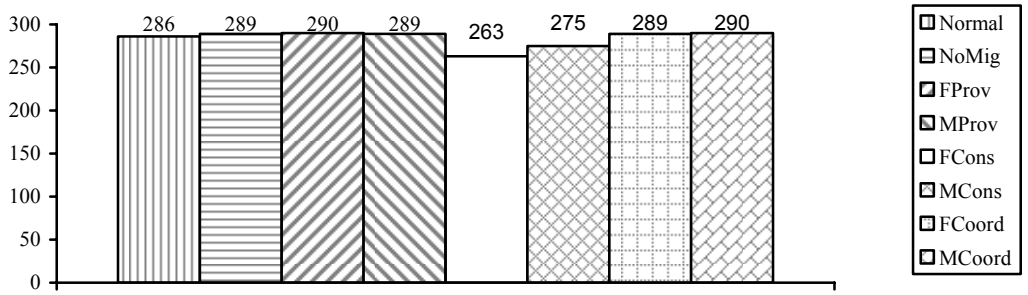
6.1.1 One Provider and One Consumer and One Coordinator

Table 6-1 1:1:1Experiment Settings

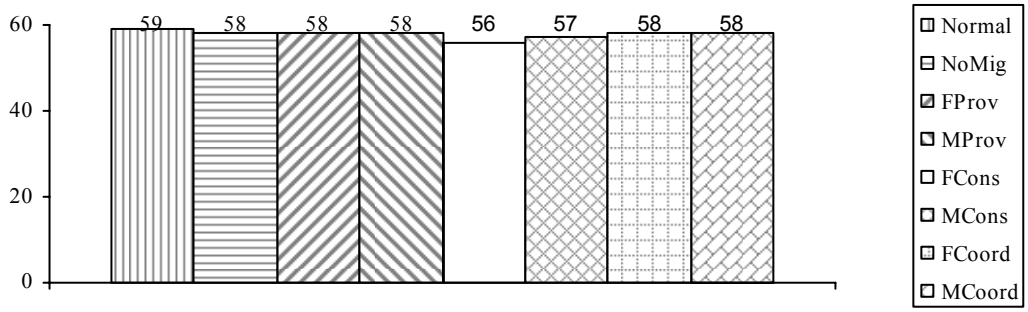
Test Case	Migration			Comment
	Times	Interval	Component	
Normal				Normal web service application without P2P-Manfold model
NoMig	0	∞	N/A	Basic P2P-Manifold application where all three components are stable
Fprov	13	~20sec	Provider	P2P-Manifold application with a frequently mobile provider
Mprov	9	~40sec	Provider	P2P-Manifold application with a mobile provider
Fcons	10	~20sec	Consumer	P2P-Manifold application with a frequently mobile consumer
Mcons	6	~40sec	Consumer	P2P-Manifold application with a mobile provider
Fcoord	12	~20sec	Coordinator	P2P-Manifold application with a frequently mobile coordinator
Mcoord	6	~40sec	Coordinator	P2P-Manifold application with a mobile provider

This situation represents the simplest P2P-Manifold application. Seven sets of experiments are performed to address the impact of the model and the status of each

component. The details of each set are listed in Table 6-1. In each experiment, no more than one component is mobile. The "Normal" test is the base case for measuring the impact of the P2P-Manifold model in different situations and the "NoMig" case is the base case for measuring the impact of the status of components.



a. Simple Service



b. Complex Service

Figure 6-1 1:1:1Service Calls in 5 Minutes

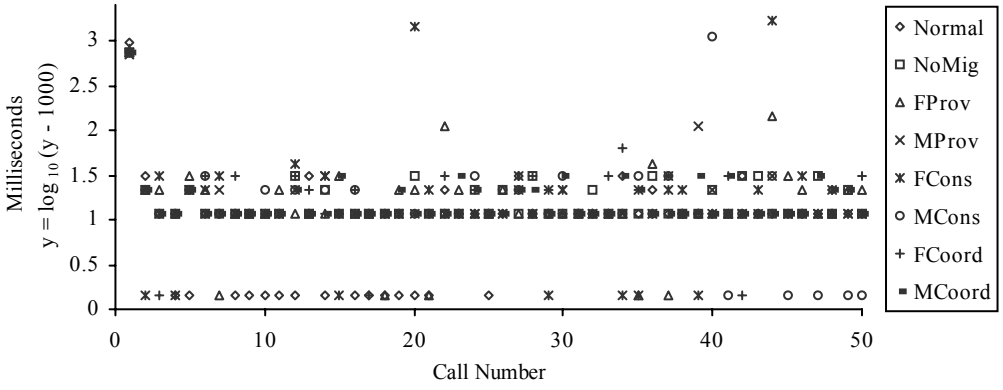
Table 6-2 1:1:1 Average Service Response Times (in millisecond, first 50 calls)

Test Case	Normal	NoMig	Fprov	Mprov	Fcons	Mcons	Fcoord	Mcoord
Simple	1029.97 (100%)	1030.08 (100.01 %)	1033.29 (100.32 %)	1030.08 (100.01 %)	1090.77 (105.90 %)	1051.31 (102.07 %)	1033.29 (100.32 %)	1030.88 (100.09 %)
Complex	5037.24 (100%)	5036.44 (99.98 %)	5057.87 (100.41 %)	5058.87 (100.43 %)	5263.57 (104.49 %)	5174.24 (102.72 %)	5040.85 (100.07 %)	5037.24 (100.00 %)

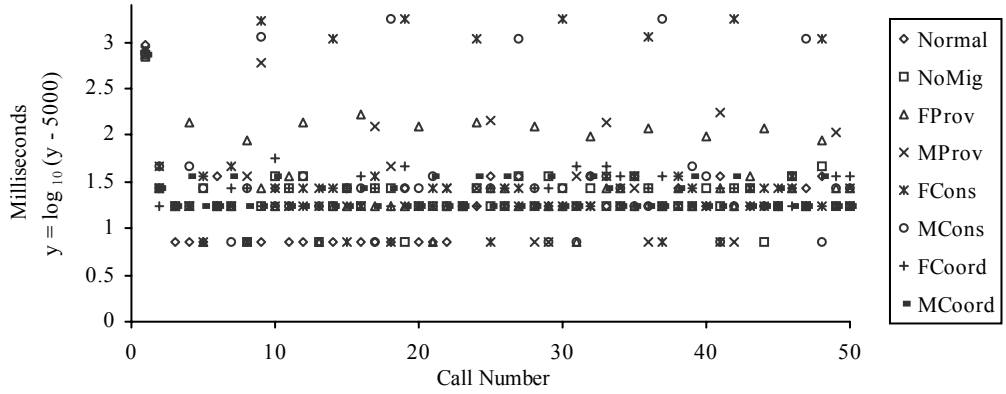
Figure 6-1 presents the throughput data of five minutes, Figure 6-2 presents the service response time of the first 50 calls and Table 6-2 summarizes the average response time of these calls. The collected data indicates:

- The P2P-Manifold model has no effect on the response time and the service throughput (comparing the test cases "Normal" and "NoMig").

- The migration of provider and coordinator has little effect on the response time and the service throughput. This is due to the caching of the local proxy, which caches the information of every-called service.
- The migration of the consumer does influence the throughput and the service response time. Compared to the "NoMig" case, the throughput of simple services decreases to 91% ±1.7% ("FCons") and 95% ±1.7% ("MCons"). This is due to the delay of first-time service calls after consumer migration. The delay is caused by the setup of the underlying channel after consumer migration while later calls benefit from the setup. Table 6-1 shows that there are 10 consumer moves in the "FCons" case and 6 moves in the "MCons" case. This suggests that the consumer should be located on a relative stable peer to increase the system performance.



a. Simple Service



b. Complex Service

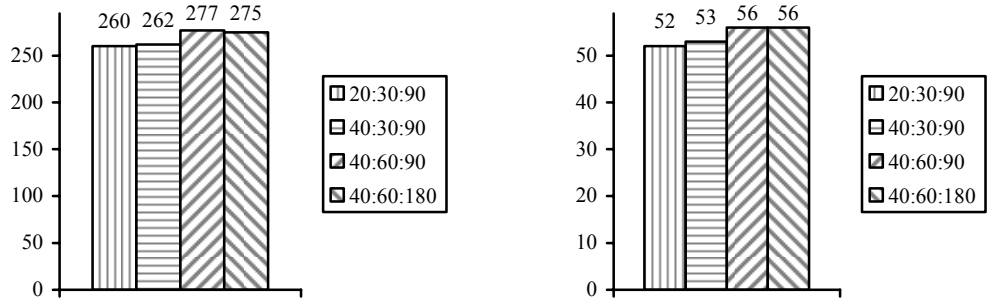
Figure 6-2 1:1:1 Service Response Time (first 50 calls)

6.1.2 One Provider and One Consumer and One Coordinator (Advanced)

This set of experiments moves the application in §6.1.1 into a P2P environment where the situation is totally dynamic and all three components are mobile. These experiments are used to address the impact of the model in a real P2P network and to evaluate the impact of the frequency of component migration. The experiment settings are listed in Table 6-3, which includes the migration rate of each component and the actual migrated time during the test period (i.e. five minutes).

Table 6-3 1:1:1 (Advanced) Experiment Settings

Migration Interval (sec) provider:consumer:coordinator	Actual Migrated Times provider:consumer:coordinator
20:30:90	13:8:3
40:30:90	7:8:3
40:60:90	7:4:3
40:60:180	7:4:1



a. Simple Service **b. Complex Service**

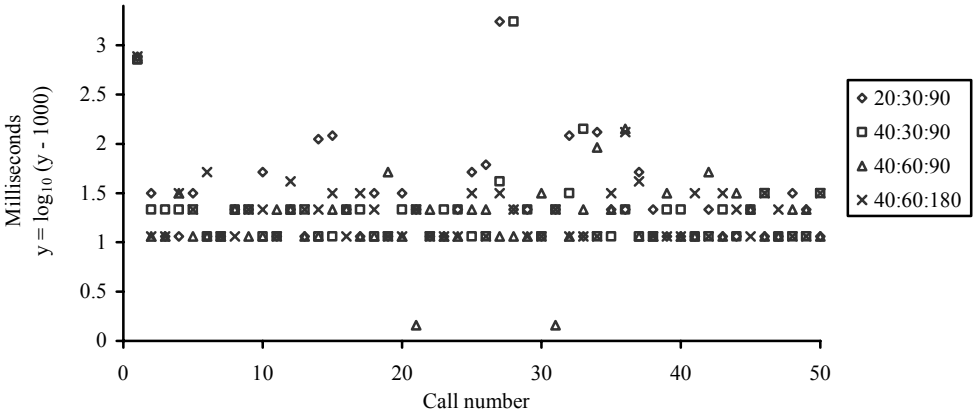
Figure 6-3 1:1:1 (Advanced) Service Calls in 5 Minutes

Table 6-4 1:1:1 (Advanced) Average Service Response Time (in milliseconds, first 50 calls)

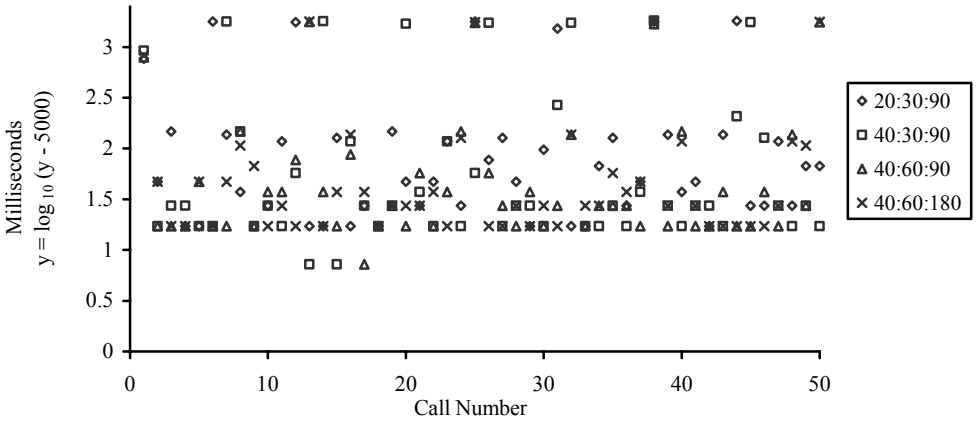
Test Case	20:30:90	40:30:90	40:60:90	40:60:180
Simple	1078.35 (100%)	1068.74 (99.11%)	1035.49 (96.03%)	1037.29 (96.19%)
Complex	5275.79 (100%)	5301.62 (100.49%)	5191.87 (98.41%)	5196.47 (98.50%)

Figure 6-3 presents the throughput of service calls in the test period. Figure 6-4 presents the response time of the first 50 calls and Table 6-4 summarizes the average response time of these calls. The collected data indicates:

- The dynamic environment reduces the throughput and increases the service latency. Compared with the result of the stable situation ("NoMig" in Figure 6-1 and Table 6-2), the throughput in this set of experiments is reduced by 4.15%±1.7%-10.03%±1.7% for simple service calls and 3.45%±1.7%-10.34%±1.7% for complex services calls.



a. Simple Service



b. Complex Service

Figure 6-4 1:1:1 (Advanced) Service Response Time (first 50 calls)

- The migration of consumers reduces both the throughput and the service response time. From the tests in this set, it can be concluded that the consumer migration rate determines the reduction of the throughput. For example Figure 6-4.b shows that the peaks of the service response time in the test cases with same consumer migration rate (i.e. "20:30:90" and "40:30:90", "40:60:90" and "40:60:180") almost overlap

each other and the throughput of those tests are similar. However, with different consumer migration rates, the throughput reduces and the service response time increases with the increase of consumer migration even if the migration rates of the other two components are kept unchanged. For example the average service response time of simple service in test case "40:30:90" (Table 6-4) is 103.21%±1.7% as that in test case "40:60:90".

6.1.3 Two Providers and Two Consumers and One Coordinator

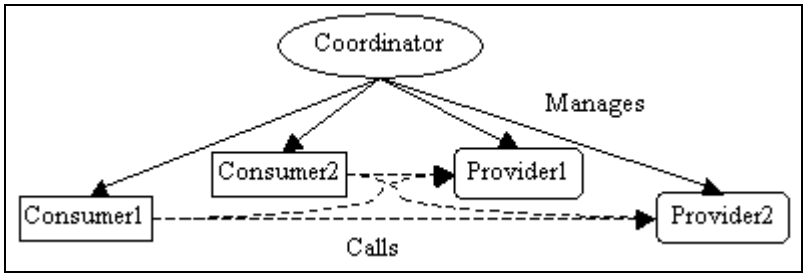


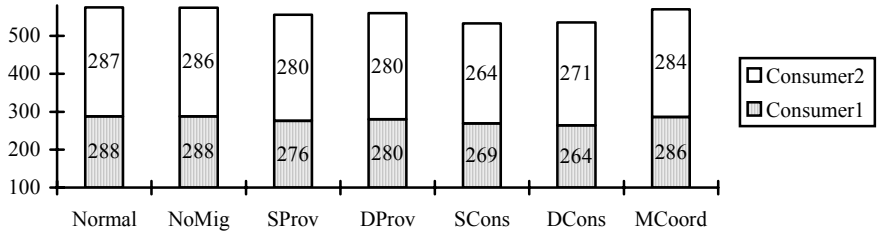
Figure 6-5 2:2:1Architecture

Table 6-5 2:2:1 Experiment Settings

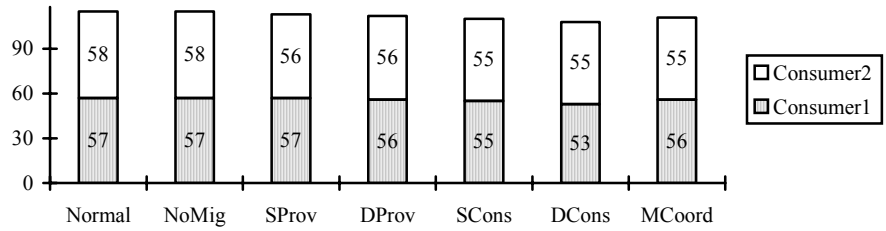
Test Case	Migration			Comment
	Times	Interval	Component	
Normal				Normal web-service application
NoMig	0	∞	N/A	P2P-Manifold application
SProv	7(Consumer1) 6(Consumer2)	40sec	Provider	P2P-Manifold application with two mobile providers who migrate simultaneously
DProv	7(Provider1) 4(Provider2)	40sec 60sec	Provider	P2P-Manifold application with two mobile providers who migrate asynchronously
SCons	6(Consumer1) 6(Consumer2)	40sec	Consumer	P2P-Manifold application with two mobile consumers who migrate simultaneously
DCons	6(Consumer1) 4(Consumer2)	40sec 60sec	Consumer	P2P-Manifold application with two mobile consumers who migrate asynchronously
MCoord	3	90sec	Coordinator	P2P-Manifold application with the coordinator migrates every interval time

This set of experiments focuses on the impact of the P2P-Manifold model with multiple consumers and providers. The test application includes two identical consumers that consume the services provided by the two providers in turns. The services provided

by the providers execute the same computation but have different names. Figure 6-5 presents the architecture of the experiment and Table 6-5 lists the settings of the experiments. In each experiment, only one type of component is mobile. If there are two components of the same type (e.g. two consumers), the components are tested with various migration rates.



a. Simple Service



b. Complex Service

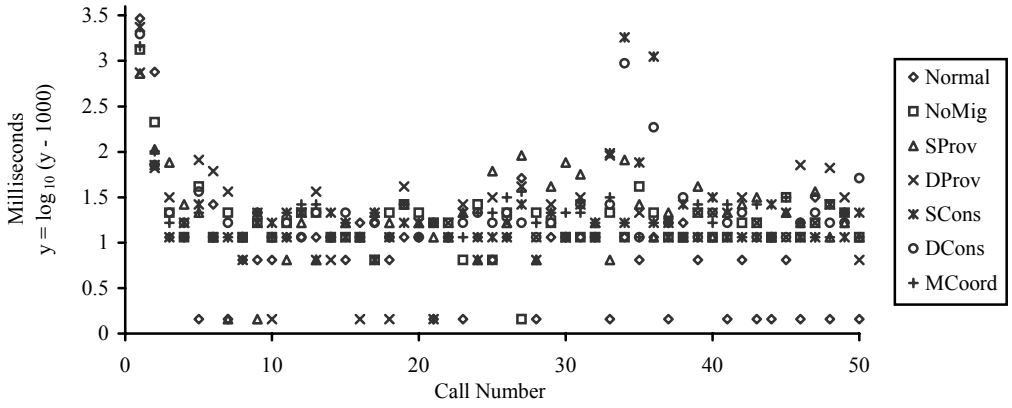
Figure 6-6 2:2:1 Service Calls in 5 Minutes

Figure 6-6 presents the throughput of the test period. Figure 6-7 presents the average response time of the first 50 calls (i.e. the average of the service response time of both consumers in order) and Table 6-6 summarizes the average response time of the calls. The collected data indicates:

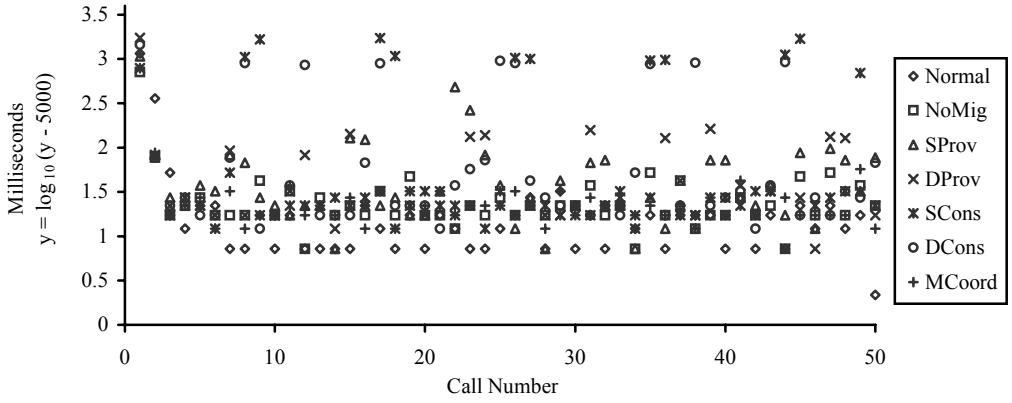
- The P2P-Manifold model has no impact on the service throughput and response time in the multiple consumers and providers situation, comparing the result of this set of experiments with those of the single consumer and provider situation (§6.1.1).
- The migration of provider and coordinator has little impact in this situation. The frequency of provider migration doesn't have impact on the throughput and the service response time. The coordinator migration influences the throughput and the response time of complex services but has no effect on the simple service.

Table 6-6 2:2:1 Average Service Response Time (in milliseconds, first 50 calls)

Test Case	Normal	NoMig	SProv	DProv	SCons	DCons	MCoord
Simple	1082.96 (100%)	1047.11 (96.69%)	1039.59 (96.00%)	1040.1 (96.04%)	1124.32 (103.82%)	1079.75 (99.70%)	1048.51 (96.82%)
Complex	5044.15 (100%)	5038.24 (99.88%)	5075.1 (100.61%)	5078.9 (100.69 %)	5295.41 (104.98%)	5198.37 (103.06 %)	5130.68 (101.72 %)



a. Simple Service



b. Complex Service

Figure 6-7 2:2:1 Average Service Response Time (first 50 calls, the average of consumer 1 and consumer2)

- The migration of consumers influences the throughput and the service response time. For example, it reduces the throughput of complex services to 96%±1.7% ("SCons") and 84%%±1.7% ("DCons") when compared with the "NoMig" case.

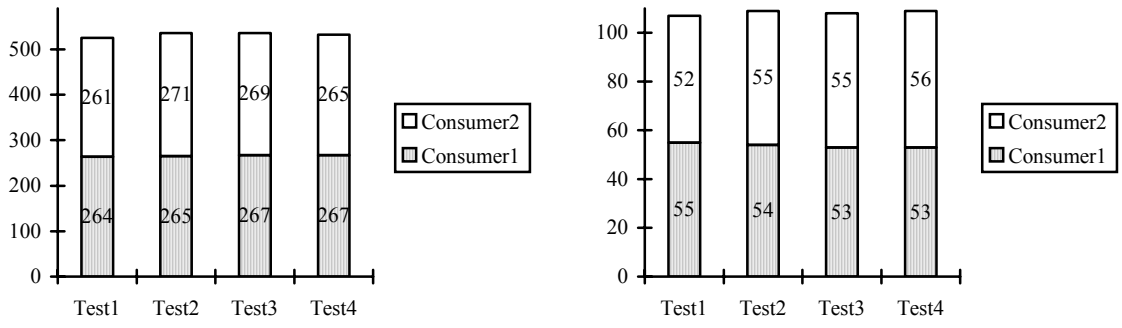
- The migration speed of consumers influences the throughput. In the test case "Dcons", the infrequent mobile consumer produced additional 2.65%±1.7% simple services and 3.78%±1.7% complex services.
- The deduction caused by consumer migration in this experiment is higher that in the single consumer and provider experiment (§6.1.1). For example, the average throughput in the test case "SCons" of this experiment is 3.09%±1.7% less than that in the test case "MCons" in §6.1.1, though the consumer(s) migrated in the same time interval. Since all communications in the model are synchronized, the extra deduction of the throughput may be the result of the waiting when two consumers ask for the same service simultaneously.

6.1.4 Two Providers and Two Consumers and One Coordinator (Advanced)

This set of experiments moves the application in §6.1.3 to a P2P environment where all components are mobile during the test period. The experiments address the impact of the P2P-Manifold model on multiple components applications in a P2P network and the impact of component migration. Table 6-7 presents the experiment settings and lists the migration rates of components. Four tests were designed to measure the impact of the frequency of each component migration on throughput and the service response time.

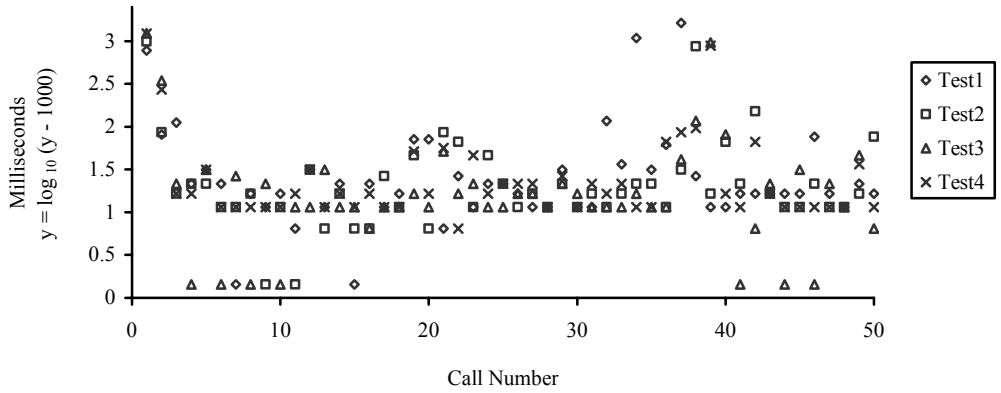
Table 6-7 2:2:1 (Advanced) Experiment Settings

	Migration	Coordinator	Consumer 1	Consumer2	Provider1	Provider2
Test1	Interval (sec)	90	40	40	30	30
	Times	3	6	6	9	8
Test2	Interval (sec)	90	40	60	30	30
	Times	3	6	4	9	8
Test3	Interval (sec)	90	40	60	30	50
	Times	3	6	4	9	5
Test4	Interval (sec)	180	40	60	30	50
	Times	1	6	4	9	5

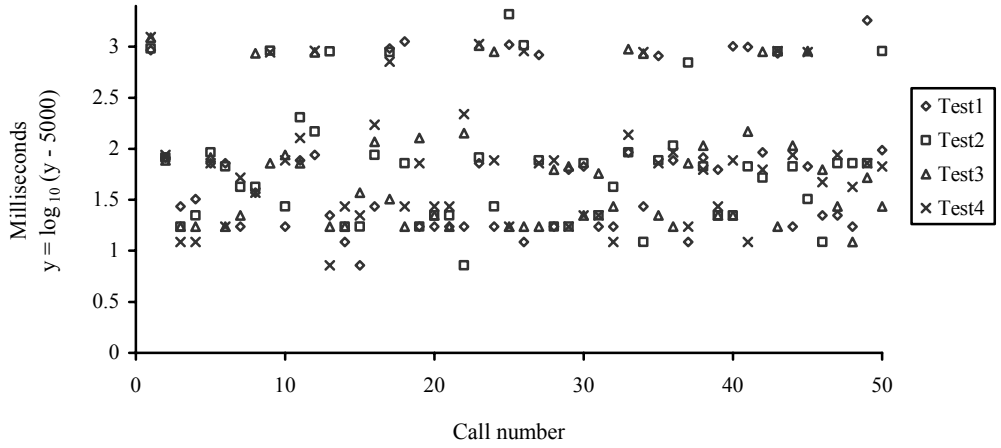


a. Simple Service **b. Complex Service**

Figure 6-8 2:2:1 Service Calls in 5 Minutes



a. Simple Service



b. Complex Service

Figure 6-9 2:2:1 Average Service Response Time
(first 50 calls, the average of consumer1 and consumer2)

Table 6-8 2:2:1 Average Service Response Time (in milliseconds, first 50 calls)

Test Case	Test1	Test2	Test3	Test4
Simple	1095.78 (100%)	1061.73 (96.89%)	1069.14 (97.57%)	1069.94 (97.64%)
Complex	5294.91 (100%)	5229.72 (98.77%)	5210.19 (98.40%)	5215.8 (98.51%)

Figure 6-8 presents the service throughput in the test period, Figure 6-9 presents the average service response time of the first 50 calls of both c`onsumers and Table 6-8 summarizes the average of the response times of the calls. The collected data indicates:

- The migration of components in this situation reduces the service throughput (Table 6-9, comparing to the stable "noMig" case in §6.1.3). The reductions in this set of experiments are in the same range as the reduction caused by the consumer migration in test cases "SCons" and "DCons". Hence it can be concluded that the migration of consumers is the most influential factor to the reduction in this situation.

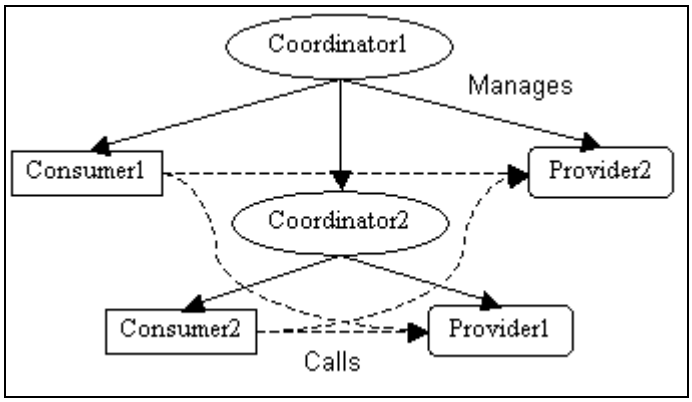
Table 6-9 Reduction of Service Throughput (base case: "NoMig" in §6.1.3, error: ±1.7%)

Test Case	Test1	Test2	Test3	Test4	SCons	DCons
Simple	8.54%	6.62%	6.62%	7.32%	7.14%	6.79%
Complex	6.96%	5.22%	6.09%	5.22%	4.35%	6.09%

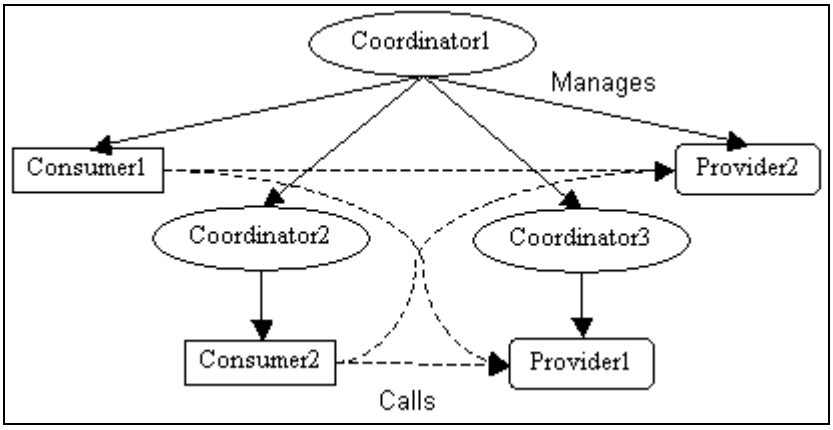
- The frequency of the consumer migration also influences the system performance. For simple service calls, the infrequent consumer migration case ("Test2") has 2% higher throughput than the frequent migration case ("Test1").

6.1.5 Organization Experiments

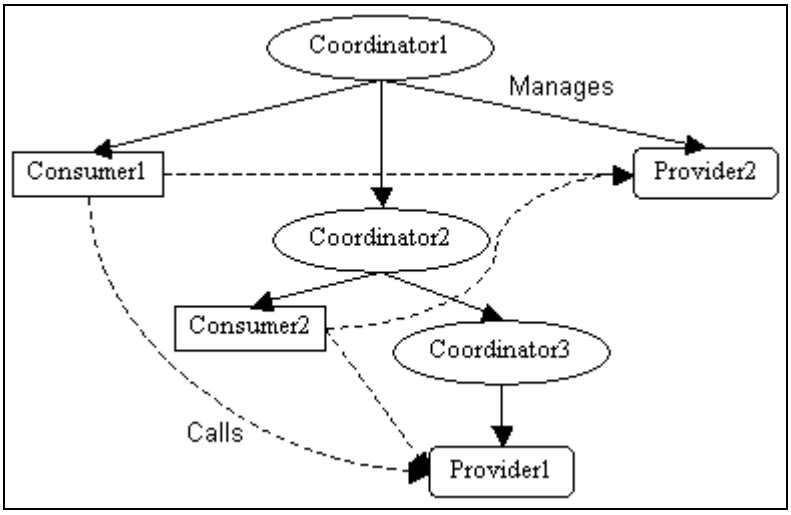
This set of experiments tests the impact of the organization in a complex situation with multiple computation components and multiple coordinators. These experiments present the impact of difference organizations for the same application as in §6.1.4 (i.e. with two consumers and two providers). The results of the experiments are useful for application optimization. This set of experiments includes four test cases: one single-coordinator situation (Figure 6-5) and three multiple-coordinator situations (Figure 6-10). Table 6-10 lists the migration settings of each component where one kind of components migrates simultaneously every predefined interval time.



Case 2



Case 3



c. Case 4

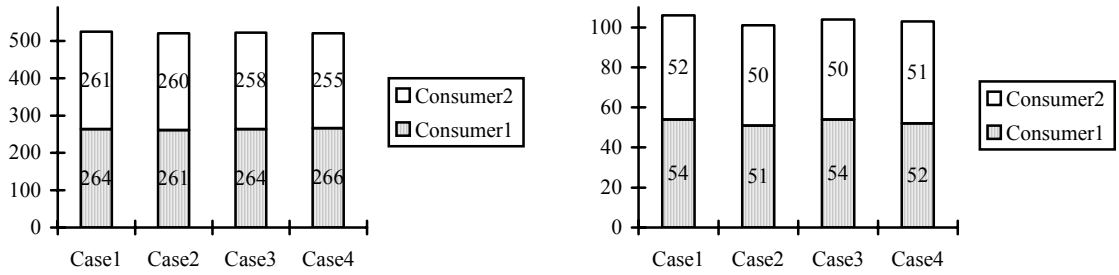
Figure 6-10 Organization Test Architectures

Table 6-10 Organization Experiment settings

Migration	Coordinator 1-3	Consumer 1	Consumer2	Provider1	Provider2
Interval (sec.)	90	40	40	30	30
Times	3	6	6	9	9

Table 6-11 Organization Experiment Service Average Response Time (in milliseconds, first 50 calls)

	Case1		Case2		Case3		Case4	
Consumer	1	2	1	2	1	2	1	2
Simple	1114.2	1078.75	1133.83	1103.39	1123.22	1118.01	1098.58	1104.39
Complex	5243.74	5343.68	5253.95	5296.62	5320.45	5344.69	5350.89	5274.78

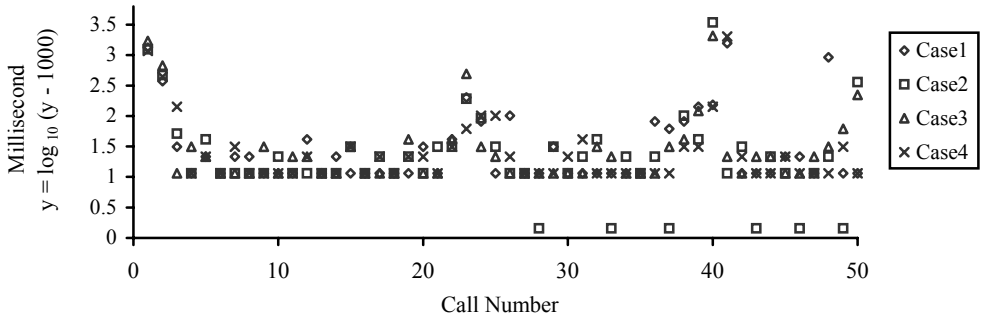


a. Simple Service **b. Complex Service**

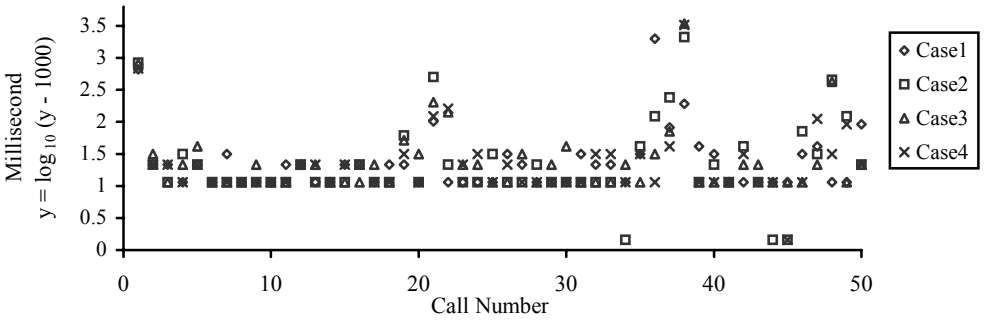
Figure 6-11 Organization Experiment Service Calls in 5 Minutes

Figure 6-11 presents the throughput of services in the test period, Figure 6-12 presents the service response time of the first 50 calls and Table 6-11 summarizes the average response time of the calls. The collected data indicates:

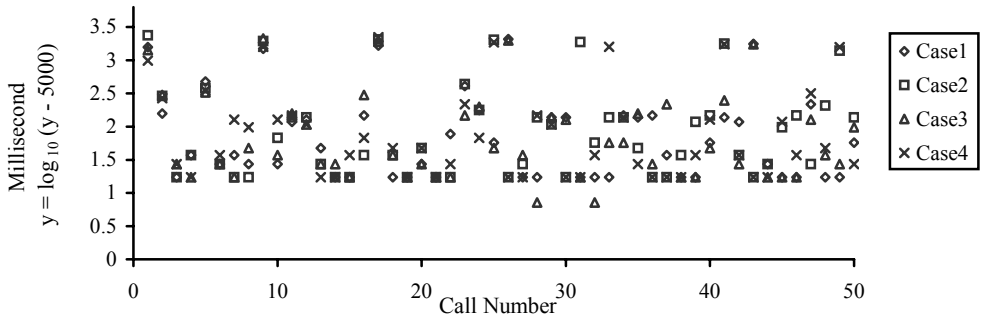
- The smaller the number of coordinators is, the more efficient the system is. The single-coordinator mode (case1) is most efficient and completed at least 0.77%±1.7% more simple calls and 4.95%±1.7% more complex calls than other three cases.
- The number of participating coordinators in cross-coordinator services affects the overall performance. Case2 and case3 are same in terms of the amount of participating mid-coordinators for cross-coordinator services. The cross-coordinator chain between Provider1 and Consumer1 in case4 needs one more participating coordinator than the other two cases and it had a bit lower throughput.



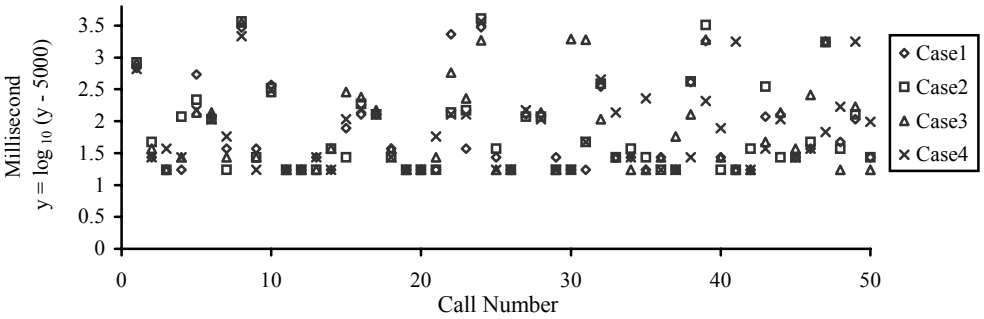
a.1 Simple Service (Consumer1)



a.2 Simple Service (Consumer2)



b.1 Complex Service (Consumer1)



b.2 Complex Service (Consumer2)

Figure 6-12 Organization Experiment Service Response Time (first 50 calls)

6.1.6 Conclusion

From the above sets of experiments, it can be concluded that:

- The migration of service providers does not influence the system efficiency.
- The migration of consumers is the most significant factor to influence the system throughput and the service response time. The more frequent consumers move, the lower performance the system has.
- The migration of a coordinator has little effect on the system efficiency. The number of participating coordinators influences the system performance as a result of the migration of midway coordinator for cross-coordinator service update.
- The organization of a workgroup influences its performance only with the amount of the participating coordinators and regardless to how the coordinators organize.
- The P2P-Manifold model itself has little effect on the system efficiency of the application while the migration of the components caused by the dynamic P2P network influence the efficiency.

6.2 System Usage Experiments

The system overhead is an important metric when measuring the system performance. Especially for P2P applications, the impact of the execution to the peer should be minimized. The following experiments monitor the system usage of weighted components of the P2P-Manifold model (i.e. local proxy and coordinator). The system usage here refers to the CPU time and the memory load.

In addition, a resource-monitoring program is developed for the Windows platforms (using DotNet). This monitor is used to measure the degree of underutilization of machines. While it is fairly common knowledge that many of the deployed computers (e.g. desktops, workstations, etc) are underutilized, it is difficult to obtain exact numbers. Organizations and individuals tend to be reluctant to publish the data due to fear of negative consequences such as unauthorized monitoring of network traffic. The resource-monitor records running processes and the usage of memory, processor and network in 10-second intervals over a period of several days.

6.2.1 System Usage Experiment

There are three kinds of components in P2P-Manifold model: coordinator, consumer and provider. A service provider is responsible for service registration and updates when performing a startup or after move. A consumer includes the consumer application(s) and a local proxy. A consumer application (without a local proxy) performs an extra service search via the custom sink to the local proxy only if it requires a service. The overhead of the provider and consumer application is minimal and can be ignored. Hence only the extra entities of the P2P-Manifold model, the local proxy and coordinator, are measured for the system usage in the experiment. For each entity, the system usage (CPU and memory) for the creation and the usage of an instance are measured.

Each test runs on two different machines five times to get the average value. Table 6-12 presents the performance data of the two test machines. In order to ignore the network traffic, all components run on the same machine.

Table 6-12 System Usage Experiment Machine Performance

Machine	CPU	RAM	O/S	Framework
Fast	Intel Pentium 4 1.8GHz	1.0GB	WindowsXP	.Net Framework 1.0
Slow	Intel Celeron 900Hz	256KB	Windows2000	.Net Framework 1.0

For each experiment, the following system data has been recorded (every 5 seconds) to measure the CPU and memory usage of each entity:

- Peak working set size (the maximum amount of physical memory that the associated component process has required all at once)
- Maximum/Minimum working set size (the maximum/minimum allowable working set size for the associated component process)
- Peak virtual memory size (the maximum amount of virtual memory that the associated component process has requested)
- Virtual memory size (the amount of virtual memory that the associated component process has requested)
- User processor time (the amount of time that the component process has spent running code inside the application portion of the process)

- Privileged process time (the amount of time that the component process has spent running code inside the operating system core)

Table 6-13 System Usage of a New Local Proxy

Machine	Working set size (MB)			Virtual memory (MB)		CPU time (sec.)	
	Peak	Max.	Min.	Peak	Total	User	Privileged
Fast	10.1	1.35	0.2	151.78	151.59	0.9	0.2
Slow	9.18	1.35	0.2	141.27	140.82	1.1	0.1

Table 6-14 System Usage of a Busy Local Proxy (in 20 seconds)

Machine	Test Case	Working set size (MB)			Virtual memory (MB)		CPU time (sec.)	
		Peak	Max.	Min.	Peak	Total	User	Privileged
Fast	1	14.00	1.35	0.2	153.99	153.81	0.92	0.22
	2	13.38	1.35	0.2	153.1	153.1	0.92	0.16
	3	13.36	1.35	0.2	154.11	153.85	0.92	0.16
Slow	1	14.31	1.35	0.2	154.31	154.13	3.04	0.28
	2	13.82	1.35	0.2	152.25	152.06	3.01	0.25
	3	14.22	1.35	0.2	153.58	153.43	3.1	0.26

Table 6-13 and Table 6-14 present the system usage of creating a new local proxy and running a local proxy respectively. Three sets of tests have been performed with a proxy to measure the impact of the workload upon the system performance: 1) with one consumer application; 2) with two consumer applications; 3) with five consumer applications. In each case, the working local proxy was tested for 20 seconds and all consumer applications are identical (searching for one of 20 services randomly every half-second). The collected data indicates:

- For setting up a new local proxy, the “fast” machine needs more memory than the “slow” machine. The total CPU time is the same for both machines.
- There is no obvious memory difference between two machines for running a busy local proxy but the “fast” machine requires less total CPU time than the “slow” machine.
- The number of associated consumer applications has no impact on either the CPU time or the memory usage.

Table 6-15 System Usage of a New Coordinator

Machine	Working set size (MB)			Virtual memory (MB)		CPU time (sec.)	
	Peak	Max.	Min.	Peak	Total	User	Privileged
Fast	10.8	1.35	0.2	146.57	146.57	0.16	0.12
Slow	11.82	1.35	0.2	145.04	145.04	1.12	0.2

Table 6-15 presents the system usage of setting up a new coordinator. The system usage of an actual working coordinator are measured by two cases: 1) one coordinator, one mobile consumer and one mobile provider; 2) two coordinators (a stable root coordinator and a mobile sub-coordinator), two consumers and two providers (Figure6-10.a). Each case ran for one minute and the system usage of the root coordinator is presented in Table 6-16. The collected data indicates:

- The memory usage for setting up a new coordinator is the same in both machines but the "slow" machine requires more CPU time than the "fast" one.
- The peak physical memory usage, the peak virtual memory usage and the total virtual memory usage increase with the increase of a coordinator's workload while the maximum/minimum working set size and the virtual memory usage are constant.
- The privileged CPU time required by the coordinator increases a little after setup (for example, on "Fast" machine, the privileged CPU time is increased only 8.3% for test 1 and 50% for test 2, Table 6-15 and Table 6-16) while user CPU time increases with the increase of workload.

Table 6-16 System Usage of Root Coordinator (in 1 minute)

Machine	Test Case	Working set size (MB)			Virtual memory (MB)		CPU time (sec.)	
		Peak	Max.	Min.	Peak	Total	User	Privileged
Fast	1	13.83	1.35	0.2	155.64	154.96	0.53	0.13
	2	14.61	1.35	0.2	155.81	154.76	0.46	0.18
Slow	1	14.09	1.35	0.2	152.58	151.71	2.16	0.22
	2	15.03	1.35	0.2	154.01	152.35	2.2	0.22

In order to measure the costs of each entity, a simple "hello-world" application (the application does nothing except printing a welcome message) is used and run on the "slow" machine. Table 6-17 presents the comparison of the system usage for local proxy, coordinator and that of the "hello-world" application. The comparison indicates that both of the entities are lightweight:

- The maximum/minimum working set size are unchanged while the peaking working set size is not more than 226% of that of the "hello-world" application.
- The virtual memory usage is no more than 125% of that of the "hello-world" application.
- The required CPU time of both entities is a bit higher than that of the "hello-world" application but is still reasonable.

Table 6-17 Comparison of System Usages

Case	Working set size (MB)			Virtual memory (MB)		CPU time (sec.)	
	Peak	Max.	Min.	Peak	Total	User	Privileged
Hello-world	6.45 (100%)	1.35	0.20	122.34 (100%)	122.34 (100%)	0.60 (100%)	0.10 (100%)
New local proxy	9.18 (142%)	1.35	0.20	141.27 (115%)	140.82 (115%)	1.1 (183%)	0.10 (100%)
Busy local proxy	14.12 (219%)	1.35	0.20	153.38 (125%)	153.21 (125%)	3.05 (508%)	0.26 (260%)
New coordinator	11.82 (183%)	1.35	0.20	145.04 (119%)	145.04 (119%)	1.12 (187%)	0.20 (200%)
Root coordinator	14.56 (226%)	1.35	0.20	153.30 (125%)	152.03 (118%)	2.18 (363%)	0.22 (220%)

6.2.2 Potential of P2P Computing

The system usage monitor runs as a non-invasive system-tray process and writes the system usage data into a local log file. This experiment was performed using some of the desktop machines in the Lab for Mobile and Ubiquitous Computing (MADMUC [41]) at the University of Saskatchewan. A total of 13 machines were involved consisting of four pool machines that are shared by all researchers, seven machines assigned to individual graduate students and two machines used by faculty members. The tests were conducted over a period of seven days (February 25th – March 4th, 2002) and a period of two days (March 26th – March 27th, 2002). The results of both tests were very similar in terms of the resource usage. Due to space limitations only the results of the first test are discussed.

The used data set is 69MB large and consists of approximately 1,572,480 records. The data set analysis indicated:

- The daily resource usage of available resources is in average below 12% (Table 6-18). In particular, the network usage is relatively low about 7%.

Table 6-18 MADMUC Lab Daily Resources Usage

Resource	CPU	Memory	Network
Average Usage	10.07%	11.32%	3.61%
Peak Usage	15.14%	12.64%	6.61%

- The usage of overall resources is medium with little fluctuations (Figure 6-13). The peak usage of resources of individual computers varies significantly per day (Figure 6-14).

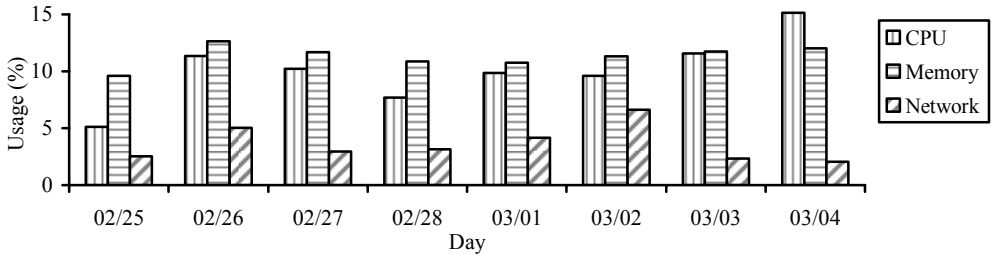


Figure 6-13 MADMUC Lab Weekly Resource Usage (%)

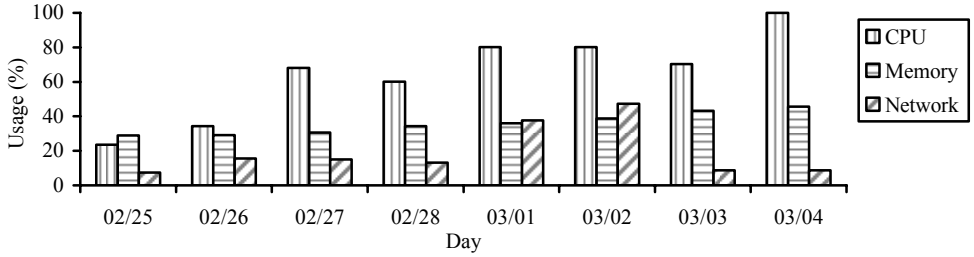
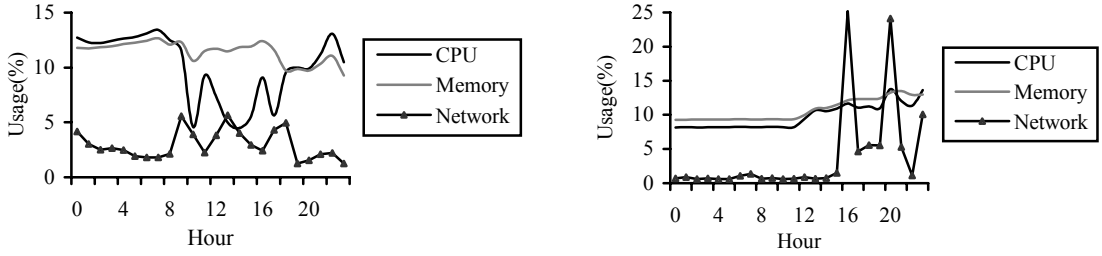


Figure 6-14 Peak Usage (%) for Individual Computer

- The daily resource usage varies from resource to resource (Figure 6-15). Below is the data for a Thursday (workday) and Saturday (free).

The above data analysis indicates that there is enough CPU, memory and most importantly network capacity available to support compute-intensive P2P applications without impacting the normal use of the machines.



Thursday 2/ 27/02 **Saturday 03/01/02**

Figure 6-15 Daily Average Usage (%)

6.3 Coordination Communication Experiments

The network overhead of the P2P-Manifold model is caused by the coordination messages. This experiment collects all coordination messages in the experiment and calculates the size of these messages to measure the network overhead. All of these recorded messages travel between managed component and coordinator and between coordinators (the local communication between consumer application and local proxy is omitted). Therefore, a coordinator-side message tracer is developed to interrupt the messages before sending out them and after receiving them.

In addition, since all messages in the experiment are in SOAP format and sent via HTTP, an extra experiment is performed to measure the speed of the SOAP/HTTP message sending. This experiment addresses the system capability and the system delay according to the message size.

6.3.1 Coordination Messages

Each communication in the model is synchronized. The coordination messages involved in the P2P-Manifold model include three types:

- **Registration message**

  A component (except for root coordinator) sends a registration message to its managing coordinator for registration after startup or migration. The managing coordinator sends registration messages to managed components after its migration. There are four kinds of registration messages: consumer registration, provider

registration, sub-coordinator registration and managing coordinator registration. The length of each registration message is fixed regardless of the difference of the component's URL value. In the throughout experiments in §6.1, the length of the request message is between 1000 bytes and 1300 bytes. The length of the response message is between 900 bytes and 1100 bytes. Figure 6-16 is an example consumer registration message.

- **Service searching message**

  A consumer sends search messages to its managing coordinator when it invokes services for the first time. The message size is fixed regardless of the difference in the service name and the provider's URL. In the experiments in §6.1, the length of search message is between 1250 bytes and 1300 bytes. However the total transferred bytes of one service search is flexible in the result of cross-coordinator service searches. The length of one cross-coordinator service search message between two coordinators is between 1250bytes to 1300bytes. Therefore using formula 2, the average total transmitted bytes of one service search requirement is:

  $$B_{service\text{-}search} = 1275 + 1275*n \quad (7)$$

  where:

  n is the amount of cross-coordinator searches.

- **Migration message**

  The transmitted messages of a component migration include the migration message that is used to keep the connection and the update message that is used to inform about the changes of the current component. In the multi-coordinator test case (Figure 6-10.a), the size of transmitted message when a root coordinator moves is:

  $$B_{root} = B_{migration} + 3*B_{suspend} + 3*B_{resume}$$
  $$= 9683 + 3*1316 + 3*1316 = 17579bytes \quad (8)$$

  where:

  $B_{migration}$ is the size of migration message.

  $B_{suspend}$ and B_{resume} are the sizes of manager suspend/resume update messages that are sent before/after the migration.

Request Message (Length: 1269Bytes)

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header>
<h4:__MethodSignature xsi:type="SOAP-ENC:methodSignature"
xmlns:h4="http://schemas.microsoft.com/clr/soap/messageProperties" SOAP-ENC:root="1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Coordination/ICoordinatoion%2C%20Version
%3D1.0.1638.33583%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
a1:COMPONENT xsd:string</h4:__MethodSignature>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<i5:Register id="ref-1"
xmlns:i5="http://schemas.microsoft.com/clr/nsassem/Coordination.Icoordinator/ICoordinatoion">
<c xsi:type="a1:COMPONENT" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Coordination
/ICoordinatoion%2C%20Version%3D1.0.1638.33583%2C%20Culture%3Dneutral%2C%20
PublicKeyToken%3Dnull">
CONSUMER</c>
<url id="ref-6">http://128.233.16.211:5123/8b5ed815_f36f_47e7_9ea4_7a7266a90f3e/
jCkmmb_vaE6_crfX8P7U6psA_2.rem</url>
</i5:Register>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Response Message (Length: 1002bytes)

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header>
<h4:__MethodSignature xsi:type="SOAP-ENC:methodSignature"
xmlns:h4="http://schemas.microsoft.com/clr/soap/messageProperties" SOAP-ENC:root="1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Coordination/ICoordinatoion%2C%20
Version%3D1.0.1638.40173%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
a1:COMPONENT xsd:string</h4:__MethodSignature>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<i5:RegisterResponse id="ref-1"
xmlns:i5="http://schemas.microsoft.com/clr/nsassem/Coordination.ICoordinator/ICoordinatoion">
<return id="ref-6">ROOTC1</return>
</i5:RegisterResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 6-16 An Example Consumer Registration Message

6.3.2 SOAP/HTTP Message Sending

This experiment tests the speed of sending SOAP/HTTP messages. The system delay of the message call is addressed by analyzing the relationship between the speed of message sending and the size of message. The test application is developed as a web service application with one provider and one consumer. Three kinds of web services are defined and tested to represent different service types:

- void Test1(byte[] bytes)
- byte[] Test3()
- byte[] Test2(byte[] bytes).

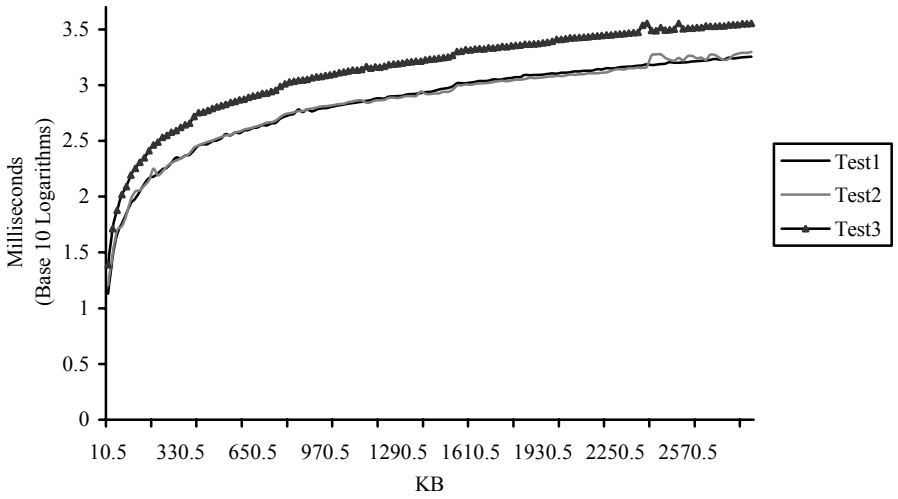


Figure 6-17 Local SOAP Message Sending Test

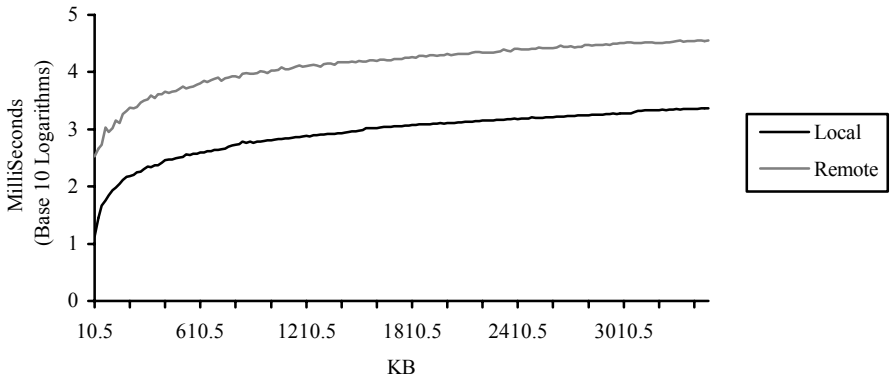


Figure 6-18 Comparison of Local Test and Remote Test (test service: Test1)

Two situations have been tested: one is local test that both components located on same machine to eliminate the network delay and one is remote test that the components were distributed. In the tests, the consumer made the service calls continuously and the size of the parameter and/or result increased by 1KB after every call. The start value of parameters and result is 1KB.

Each experiment has run five times. Figure 6-17 and Figure 6-18 presents the average of the tests. Since the data set is very large to present, the present data set is the average of every 20 calls and only the calls less than 5700KB are considered. The collected data indicates:

- The service response time linearly increases with the increase of the message size. The response time of the tests (Figure 6-18) can be summarized as follows:

  $ResponseTime_{local}$ (millisecond) ≈ 0.66 * Message size (KB)

  $ResponseTime_{remote}$ (millisecond) ≈ 10.48 * Message size (KB).

- In the local tests (Figure 6-17), the service response time of Test1 and Test2 are almost same while the response time of Test3 equals nearly the sum of Test1 and Test2. It indicates that in the .NET Framework the speed of message passing depends mostly on the message size. Disregarding the service computation time, the service response time is irrelative to the service signature. The SOAP-format communication can be measured by the transmitted bytes and ignoring the parsing delays.
- The maximum acceptable size of a SOAP message is between 5459KB and 5460KB. The maximum size of one message collected from experiments in §6.1 is less than 10KB and it is far below to the maximum acceptable size. Therefore though SOAP is heavier than the binary format, it is still acceptable.

6.4 Summary

The above three sets of experiments measure the P2P-Manifold model in several aspects: the system throughput, the system overhead and the network overhead. The system throughput is measured in several different situations to address the impact of

the model and the component migration. The system overhead of the model is measured by the cost of the extra components and how it influences the working peer. The network overhead is measured by analyzing the size and the sending speed of the coordination messages. From the analysis, it can be concluded that:

- The P2P-Manifold model has little impact on the efficiency of application. The migration of components due to the dynamic P2P network influences the throughput.
- The organization of the workgroup influences the system throughput where the number of participating coordinators is much more affective than the organization of these coordinators.
- The P2P-Manifold model is lightweight in terms of the system overhead caused by the model. Through the potential of P2P computing experiment, there are plenty of unutilized system resources (e.g. CPU, memory and network bandwidth) to support compute-intensive P2P applications without impacting the normal use of the machines.
- The network overhead caused by the model is reasonable and won't cause network traffic. Coordinator migration is costly in terms of the network traffic.
- The platform-independent SOAP message format won't cause capability problems.

CHAPTER 7

SUMMARY & CONTRIBUTIONS

7.1 Summary

P2P networks are a recent addition to the already large number of distributed system models. Unlike other forms of computing, P2P still lacks of research especially in the area of coordination. On the other hand, coordination has been one important issue in the area of distributed computing for a long time and a number of models and languages have been developed.

This research focuses on applying an existing coordination model to meet the requirements of the P2P computing environment. The proposed P2P-Manifold model is an extension of the existing event-driven Manifold coordination model and implements the transparency of the component migration. This thesis presents the design and implementation of this model and its evaluation.

The P2P-Manifold model uses m-ary trees to organize two kinds of components: coordinator and computation components. A coordinator is a component dealing with the connections between computation components and computation components are designed as back-box and deal only with the computational services.

The model is evaluated for: system overhead, throughput, programmability and component availability. The P2P-Manifold model is suitable for developing P2P computing applications:

- Availability - all services in a workgroup are highly available during the lifetime of the execution.
- Efficiency - the P2P-Manifold model has little impact to the throughput and service response time of application.
- Lightweight - the system usage (CPU and memory) of the extra coordination-oriented components of the model (i.e. local proxy and coordinator) is low.

- Economy - the network overhead caused by the coordination message is affordable and won't result in network traffic.
- Flexibility - the model is platform-independent and language-independent due to the usage of standard protocols (e.g. SOAP).
- Transparency - the model hides the migration of component in the result P2P applications. Using this model, P2P application are developed and viewed as normal distributed computing applications.
- Programmability - embedding the model into existing middleware frees the developers from the complicated coordination concepts.

7.2 Contributions

By designing, implementing and evaluating the P2P-Manifold model, this thesis provides the following contributions:

- Introduction of the coordination issues of P2P computing, in particular the importance of the transparent component migration – increasing the flexibility and programmability of P2P systems.
- Design and implementation of a new P2P coordination model, P2P-Manifold. The model provides a transparent P2P environment by wrapping the component migration and hiding the complicated coordination efforts from application developing.
- Evaluation of the P2P-Mainfold model with three essential factors: system throughput, system overhead and network overhead. Through the experiments, the model is proven to be productive and feasible for P2P application development.

CHAPTER 8

FUTURE WORK

This research designs and implements the basic coordination model for P2P computing. Therefore the future work will focus on:

- **Context-dependent service support**

  A context-dependent service is useful when developing long-lived services with state. The support requires complex provider migration instead of the simple one for context-independent service. In order to keep consistency in the system, the migration of context-independent service requires suspending the service. The impact of the provider's migration is an important aspect when evaluating the model.

- **Service duplication**

  In the P2P-Manifold model, one service is provided by only one provider. This simplifies the service searching but is inefficient for busy services. Since the P2P network is assumed with plenty of idle resources, it is possible to duplicate busy services so as to share the workload and speed up the service response time.

- **Service caching**

  The coordinator in the model is assumed relative stable, while the computation component may move more often. The frequent migration of computation components causes the loss of service results and the temporary loss of services. This problem reduces the throughput of applications by increasing the amount of invalid service invocation. A coordinator can cache the service result when one consumer is moving and resend it after it resumes. In addition, the cached result can serve other consumers if the same service is called.

- **Others**

  This research is based on several assumptions such as plentiful resources and autonomous peer selection (§1.2). However all of these assumptions may break. The model needs to be extended to meet the situation when one or some assumptions are untrue.

LIST OF REFERENCES

[1] Milojicic D., Kalogeraki V., Lukose R., Nagaraja K., Pruyne J., Richard B., Rollins S., Xu Z., and HP Laboratories Palo Alto, "*Peer-to-Peer Computing*". Technical Report, 2002.

[2] "*BitTorrent*". August 2004. http://bitconjurer.org/BitTorrent/

[3] Clip2, "*The Gnutella Protocol Specification v0.4 (Document Revision 1.2)*". 2001.

[4] Clarke I., Sandberg O., Wiley B., and Tong T., "*Freenet: A Distributed Anonymous Information Storage and Retrieval System*". Designing Privacy Enhancing Technologies, 2000, 2009: 46-66.

[5] "*SETI@home*". August 2004. http://setiathome.ssl.berkeley.edu

[6] Papadopoulos G. and Arbab F., "*Coordination Models and Languages*". Advances in Computers (volume 46), Academic Press, 1998, pp. 329-400.

[7] "*FastTrack*". August 2004. http://www.fasttrack.net

[8] "*Jabber Software Foundation*". August 2004. http://www.jabber.org/

[9] "*Zoogi*". August 2004. http://www.zoogi.com

[10] "*Project JXTA*". August 2004. http://www.jxta.org

[11] "*Groove Networks*". August 2004. http://www.groove.net/

[12] Barkai D., "*An Introduction to Peer-to-Peer Computing*". Technical Paper, Intel Developer Update Magazine, 2001.

[13] O'Reilly T., "*ICQ as a P2P Pioneer*", Technical Paper, O'Reilly Network, 2001.

[14] Condor Team, "*An Overview of the Condor System*". Technology Overview, University of Wisconsin-Madison, Madison USA, 2004.

[15] "*CondorView Pool Statistics*". 2004. http://pumori.cs.wisc.edu/condor-view-applet

[16] "*Avaki*". August 2004. http://www.avaki.com

[17] Avaki Corporation, "*Avaki Grid Software: Concepts and Architecture*", White Paper, 2002.

[18] Sun Microsystems, Inc., "*Project JXTA: An Open, Innovative Collaboration*". Technical Paper, 2001.

[19] Anthill Team, "*The Anthill Project*". Technical Report, Dept. of Computer Science, University of Bologna, Bologna Italy, 2001.

[20] Matos G. and Purtilo J., "*Reconfiguration of Hierarchical Tuple-spaces: Experiments with Linda-polylith*". Technical report, University of Maryland, Adelphi USA, 1993, UMIACS-TR-93-100.

[21] Kirtland M., "*A Platform for Web Services*". Technical Paper, Microsoft Developer Network, 2001.

[22] Microsoft .NET Framework Developer Center, "*.NET Framework Technology Overview*". Technology Overview, 2004.

[23] Kirtland M., "*A Platform for Web Services*". Technical Paper, Microsoft Developer Network, 2001.

[24] W3C, "*Extensible Markup Language (XML) 1.0 (Third Edition)*". Recommendation, August 2004. http://www.w3.org/TR/REC-xml

[25] W3C, "*SOAP Version 1.2 Part 0: Primer*". Recommendation, 2003. http://www.w3.org/TR/soap12-part0

[26] W3C, "*Web Services Description Language (WSDL) 1.1*". Note, 2001. http://www.w3.org/TR/wsdl

[27] "*uddi*". August 2004. http://www.uddi.org/

[28] Carriero N. and Gelernter D., "*Coordination Languages and Their Significance*". Communications of the ACM, ACM Press, 1992, 35(2): 97-107.

[29] Schumacher M., "*Designing and Implementing Objective Coordination in Multi-Agent Systems*". Ph.D. Thesis, Dept. of Informatics, University of Fribourg, 2000.

[30] Arbab F., Ciancarini P., and Hankin C., "*Coordination Languages for parallel Programming*". Parallel Computing, Elsevier Science Publishers B. V., 1998, 24(7): 989-1004.

[31] Roman G. and Cunningham H., "*Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency*". IEEE Transactions on Software Engineering, IEEE Press, 1990, 16(12): 1361-1373.

[32] Sun Microsystems, Inc., "*JavaSpaces™ Specification*". Specification, 1998.

[33] "*JINI.org*". August 2004. http://www.jini.org/

[34] Arbab F., "*Manifold Version 2.0*", User reference, CWI, The Netherlands, 1998.

[35] Magee J., Dulay N., and Kramer J., "*Structuring Parallel and Distributed Programs*". Software Engineering Journal, 1993, 8(2): 73-82.

[36] Dulay N. "*A Configuration Language for Distributed Programming*", Ph.D. Thesis, Dept. of Computing, Imperial College, London UK, 1990.

[37] Magee J., Dulay N., Eisenbach S., and Kramer J., "*Specifying Distributed Software Architectures*". Proceedings of the 5th European Software Engineering Conference, London, UK, 1995, pp. 137-153.

[38] Vlassis N., "*A concise Introduction to Multiagent Systems and Distributed AI*". Informatics Institute, University of Amsterdam, Amsterdam, Netherlands, 2003.

[39] Krone O., Chantemargue F., Dagaeff T., and Schumacher M., "*Coordinating Autonomous Entities with STL*". ACM SIGAPP Applied Computing Review, ACM Press, 1998, 6(2): 18-32.

[40] "*The Darwin Language, Third Version*". User Reference, Dept. of Computing, Imperial College of Science, Technology and Medicine, London UK, 1992.

[41] "*MADMUC Lab*". August 2004. http://bistrica.usask.ca/madmuc

[42] Obermeyer P. and Hawkins J., "*Microsoft .NET Remoting: A Technical Overview*". Technical Paper, Microsoft Corporation, 2001.

[43] Srinivasan P., "*An Introduction to Microsoft .NET Remoting Framework*". Technical Paper, Microsoft Corporation, 2001.

[44] Foster I., "*The Grid: A New Infrastructure for 21st Century Science*". Physics Today, 2002, 55(2): 42-47.

APPENDIX A

.NET REMOTING

The Microsoft .NET Remoting Framework allows objects in different applications and different machines to communicate with each other via XML messages. The framework provides a number of services, including activation and lifetime support. It also provides communication channels responsible for transporting messages to and from remote applications [42].

A.1 Architecture

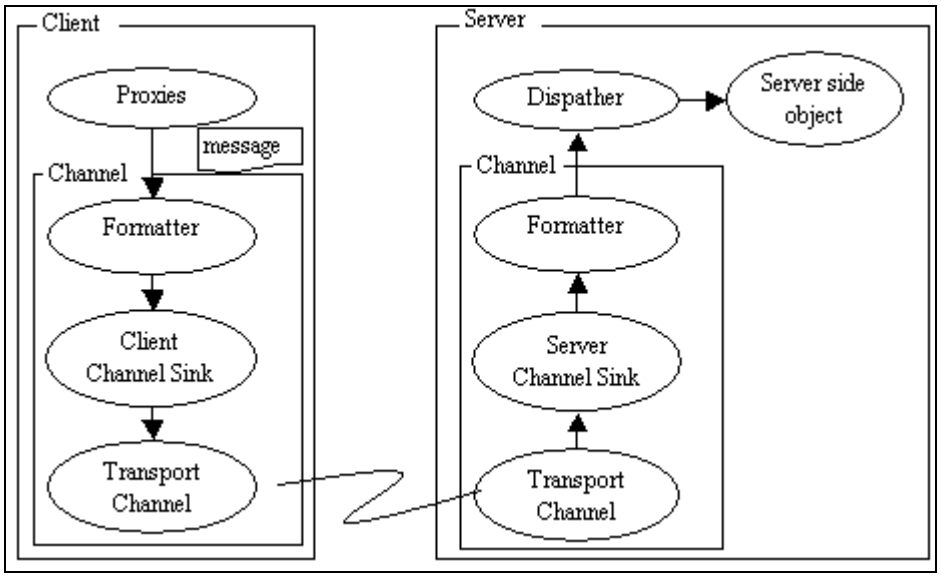


Figure A-1 .NET Remoting Architecture

The .NET *Channel Services* provides the underlying transport mechanism for transport messages to and from remote objects. When a client calls a method of a remote object, the parameters, as well as the call data, are transported through the channel chain to the remote object. Any results from the call are returned back to the client in the same way.

The .NET Remoting is a layered architecture (Figure A-1) and consists of six core object types:

- **Proxies**

  These objects masquerade as remote object and ensure that all calls made are forwarded to the correct remote object instance. When a client activates a remote object, the framework creates a local instance of the class *TransparentProxy* that provides the same interface as the target object. Then, the proxy creates an instance of the specified channel object and begins traversing its sink chain.

- **Messages**

  Message objects contain the data to execute a remote method call. The .NET Framework defines several special types of messages. A *ConstructionCall* message is used during the instantiation of CAOs. A *MethodCall* message and its respective return message represent the method call request and response.

- **Channel sinks**

  The channel chain contains the sinks required for basic channel functionality. It normally has at least two standard sinks that begin and end the chain: the formatter sink and the transportation sink. In between the two, programmers can define as many custom sinks as needed. These sink objects allow custom processing of message during a remote invocation. Sinks read or write data to the stream and add additional information to the headers where desired.

- **Formatter**

  The formatter serializes/desterilizes the message into/from a transfer format. There are two native formatters in the .NET runtime, namely Binary and SOAP. Other implementations can use their own means to transform the channel message into the stream.

- **Transport channel**

  The transport channel is responsible for sending and retrieving message between the client and the server. The .NET Framework supplies the HTTP and TCP channels but third parties can write and plug in their own channels. The HTTP channel use

SOAP by default to communicate, whereas the TCP channel uses Binary payload by default.

- **Dispatcher**

  The dispatcher in the server side takes the decoded message and forwards the method call to the real destination object for processing.

A.2 .NET Remoting Objects

The remote service (object) in .NET Remoting can be accessed in two ways. The first technique is referred to as marshal-by-value (MBV). It makes a full copy of the remote server on the local machine for accessing it locally.

The other possibility is known as marshal-by-reference (MBR). In the latter approach, each server object has an interface for all exposed methods. Clients use this interface to make a local transparent proxy, which make the remote call as if it is a local call to the application. There are three types of objects that can serve as .NET remote objects [40]:

- **Single call**

  A server (service provider) creates a single call object every time when a service request coming in. Single call objects cannot hold state information between method calls. However, single call objects can be configured in a load-balanced fashion.

- **Singleton objects**

  Only one instance of a singleton object can exist at any given time. Those objects share data by storing state information between client invocations. Both single call objects and singleton objects are Server-activated object (SAO).

- **Client-activated objects (CAO)**

  Client-activated objects (CAO) are server-side objects that are activated upon request from the client. When the client submits a construction request for a server object, an activation request message is sent to the remote application. The server creates an instance of the requested class and returns an object reference back to the client application that invoked it. A proxy is then created on the client side using the

object reference. The client's method calls will be executed on the proxy. Each activation invocation returns a proxy to an independent instance of the server type.

The object's lifetime in .NET Remoting is managed by a time-to-live (TTL) counter. For an object that has object references transported outside the application, a lease is created. The lease-based concept assigns a TTL counter to each remote object created at the server. As soon as the time counter reaches zero, the lease expires and the object is marked as timed out and ready for garbage collection. The time is incremented when method call placed on the remote object.

A.3 Configuration Files

To configure the .NET Remoting application, one can choose to either hard-code all channels and objects or use the standard .NET Remoting configuration file. The configuration files separate the configuration information from the client code. Future changes can be made through configuration file changes without the need to recode or recompile.

The .NET Remoting configuration files are XML documents. A typical configuration file includes the following information [43]:

- Host application information
- Name of the objects
- URI of the objects
- Channels being registered
- Lease Time information for Server Objects

Figure A-2 is an example of a configuration file for server-side application. It informs the .NET Remoting server application to publish a SAO object, whose full type name (type, assembly) is "Foo, common". The URI of the remote object is "Foo.soap" and it is accessible through HTTP channel at port 9000.

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown type="Foo, common" objectUri="Foo.soap" mode="Singleton" />
         </service>
         <channels>
            <channel ref="http" port="9000" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Figure A-2 Server-Side Configuration File

A.4 Comparing With Other Distributed Object Paradigms

Before .NET Remoting, there were several underlying technologies of choice for remote object communication. Most of these paradigms try to hide the location difference of server and client and make the remote method invocation look like a local call. Two of the most popular distributed object paradigms are [44]:

- **JavaSoft's RMI (Remote Method Invocation)**

  RMI relies on a protocol called the Java Remote Method Protocol (JRMP). Both the RMI server object and the client object have to be written in Java. The server object exposes the methods in an interface and is hold by the RMIRegistry that runs on the Server machine. A client acquires an object reference to a server object by doing a lookup.

- **OMG's CORBA (Common Object Request Broker Architecture)**

  Each server object in CORBA exposes their methods to the central object bus, Object Request Broker (ORB)**.** CORBA uses a protocol called IIOP (Internet Inter-ORB Protocol) to communicate between different systems.

Transparent invocation is the core idea of .NET Remoting and the above technologies. The transparency is realized by the client side stub/proxy. The stub/proxy is created using the server object interface.

Compared to .NET Remoting, the above two architectures have some problems. The main problem is that these protocols are non-semantic and hence incompatible. The communication between heterogeneous components can only happen via a bridge.

Another problem is that these protocols are not firewall friendly. Most firewalls are configured to allow access only through specific ports, the most popular being the HTTP port 80. Those protocols use different ports and stick on the unitary message format, mostly binary format, which are blocked by most corporate firewalls.

A.5 Benefits for Distributed Application Development

.NET Remoting framework is an integration of .NET Framework and web services concepts. It harvests the benefits of both of them:

- **Multi-language support and integration**

  .NET Framework supports many languages such as C#, C, VB, Fortran, etc. A developer can choose any familiar or suitable language for application development.

- **Security**

  Microsoft® .NET Passport is a core component of the Microsoft .NET initiative. The online service (Passport) enables authentication of users. Once authenticated, the user can roam across passport-participating Web sites.

- **Semantic message**

  The standard protocols of web services, SOAP, UDDI and WSDL, are XML protocols. The request and response messages of method calls are both machine readable (using XML parser) and human legible. The metadata (tags) in XML document helps the understanding of data's purpose and use.

- **Platform and language independence**

  Web services are internet-ready and are an open standard ratified by the W3C (The World Wide Web Consortium). With SOAP and HTTP, it is possible to communicate with heterogeneous program written in any programming language on almost all the major computing platforms.

- **Cross firewall**

  Web services can use HTTP transport protocol or other Internet-friendly protocols (such as SMTP). Web services with XML data encoding and HTTP protocol is firewall friendly.

APPENDIX B

.NET P2P-MANIFOLD IMPLEMENTATION

The .NET P2P-Manifold is implemented in C# on .NET Framework 1.1 as linked libraries. Figure B-1 presents the class diagram of this implementation.

B.1 Libraries

The P2P-Manifold implementation includes six linked libraries and an application:

- **MySink.dll**

  The *MySink* library defines the client sink, which must be included in any consumer application to interrupt and redirect outgoing service. It includes two classes *ClientSink* and *ClientSinkProvider*.

- **IProxy.dll**

  The *IProxy* library is the shared library, which declares the web service interface of local proxy object. It includes:

  - *IComponent* interface that is the base interface and must be implemented by all non-black-box component of the model (i.e. coordinator and local proxy);
  - *IProxy* interface that declares the interface of local proxy object;
  - *IRepository* interface that declares the basic methods of the cache table of local proxy and *HashtableRep* class is an implementation of hashtable-style repository;
  - *ProxySuspendedException* that is an exception class and defines the exception rose when an operation is made to a suspended proxy.
  - *STATUS* enumeration that lists out all possible statuses of a component.

- **ICoordination.dll**

  The *ICoordination* library includes *ICoordiator* interface, which declares the web service interface of a coordinator and supplement definitions: struct *KeyElements*

(state information), enumeration *COMPONENT* (all component types) and *Manager* class (managing coordinator representative).

- **Coodination.dll**

  The *Coordination* library includes the *Coordinator* class (the implementation of coordinator entity) and the representatives of all managed components: *Consumer* (for managed consumer), *Provider* (for managed provider), *SubCoordinator* (for managed coordinator), *PseudoConsume*r (for logical consumer) and *PseudoProvider* (for logical provider). All these components classes inherit from the base (abstract) class *Component* and are contained in a *ComponentDB* instance.

- **LocalProxy.dll**

  The *LocalProxy* library implements the required functions of a local proxy within three entities: *ILocalProxy* interface defines the non-web-services methods, which can be called by inter-object invocation; the *Proxy* class represents local proxy and implements its functions; the *LocalProxy* class provides the functionality to create a new local proxy.

- **RegAssist.dll**

  The *RegAssist* library defines the help methods for setting up applications and registering/updating services. It also defines the base (abstract) class *Services*, which provides service update methods and must be inherited by all service classes of provider.

- **Coordinator.exe**

  It is the application to run a coordinator. A configure file is required to help set up the coordinator.

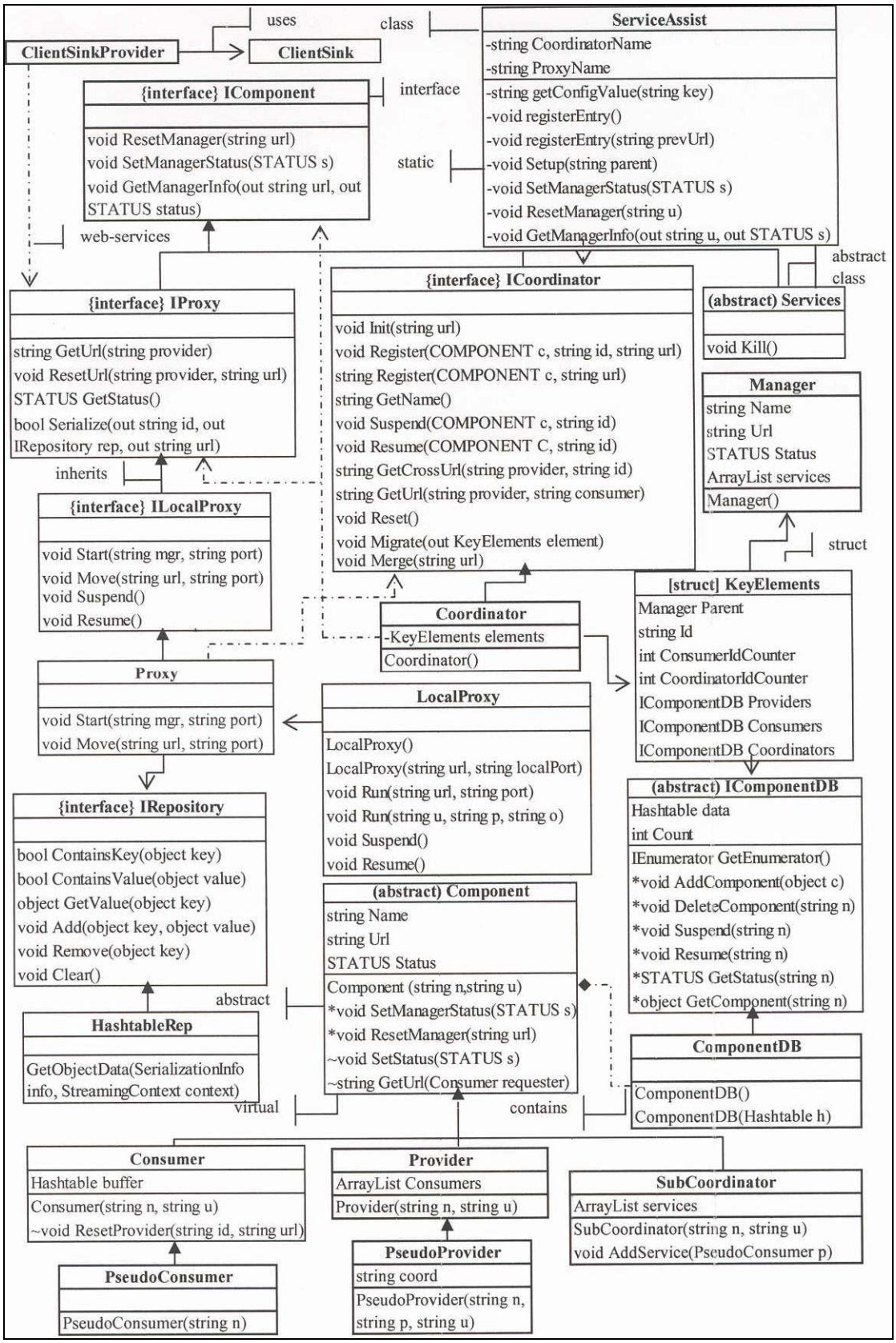


Figure B-1 .NET P2P-Manifold Class Diagram

B.2 Classes

- **ClientSink**

  Implements required functions for the custom channel sink, which is involved in the channel of consumer application to interrupt and redirect outgoing service call.

  Base Classes: *BaseChannelSinkWithProperties*

  Implemented interfaces: *IClientChannelSink*

  Type: class

- **ClientSinkProvider**

  Creates the custom channel sink for the consumer-side channel through which remoting messages flow.

  Implemented interfaces: *IClientChannelSinkProvider*

  Type: class

- **Component**

  Provides required functions for component proxies, which will be stored in the coordinator's component database to represent the real remote component.

  Type: abstract class

  Attributes: *Serializable*

  Properties:

  - *Name* – a *String* instance stores component name
  - *Status* – a *STATUS* instance stores component status. There are three statuses defined: READY, SUSPEND, DONE.
  - *Url* – a *String* instance stores component URL

  Functions:

  - *GetUrl* – virtual method, gets the URL of the represented component
  - *ResetManager* – abstract method, resets the URL of the managing coordinator
  - *SetManagerStatus* – abstract method, resets the status of the managing coordinator
  - *SetStatus* – virtual method, resets the status of the represented component

- **CompomentDB**

  Implements a synchronized hashtable-style (key: component name) container for coordinator to contain component proxies.

  Type: class

  Attributes: *Serializable*

- **Consumer**

  Represents a remote consumer component.

  Base Classes: *Component*

  Type: class

  Attributes: *Serializable*

  Properties:

  - *buffer* – a *Hashtable* instances that works as a temporary message cache when the consumer suspends. The cache will be emptied by sending out all messages when the consumer resumes.

  Functions:

  - *ResetProvider* – virtual method, updates the provider information.

- **Coordinator**

  Provides functions and properties of a coordinator component.

  Base Classes: *MarshalByRefObject*

  Implemented interfaces: *ICoordinator*

  Type: class

  Properties:

  - *KeyElements* – a *KeyElements* instance stores all state information, which needs to be cloned to keep continuity when coordinator moves.

- **HashtableRep**

  Implements a hashtable-style repository for local proxy to cache the information of ever-used providers.

  Implemented interfaces: *IRepository*

  Type: class

Attributes: *Serializable*

- **IComponent**

  Provides the syntax of required web-service functions of component instances (i.e. provider, consumer and coordinator).

  Type: interface

  Functions:

  - *GetManagerInfo* – gets the information (i.e. URL, status) of the managing coordinator.
  - *ResetManager* – resets the URL of the managing coordinator.
  - *SetManagerStatus* –resets the status of the managing coordinator.

- **IComponentDB**

  Provides the syntax of required functions and properties of a coordinator's component database instance.

  Type: abstract class

  Properties:

  - *Count* – an *Integer* instance stores the count of stored components.
  - *data* – a *Hashtable* instance keeps the component objects.

  Functions:

  - *AddComponent* – abstract method, adds a new component to the database.
  - *DeleteComponent* – abstract method, deletes a component by the given name.
  - *GetComponent* – abstract method, gets the component by the given name.
  - *GetEnumerator* – returns an *IEnumerator* for the database.
  - *GetStatus* – abstract method, gets the status of a stored component identified by the given name.
  - *Resume* – abstract method, resumes a stored component identified by the given name.
  - *Suspend* – abstract method, suspends a stored component identified by the given name.

- **ICoordinator**

  Provides the syntax of additional required web-service and non-web-service functions of a coordinator instance.

  Implemented interfaces: *IComponent*

  Type: interface

  Functions:

  - *GetCrossUrl* – searches for a cross-coordinator service and returns the URL of the provider.
  - *GetName* – gets the name of current coordinator.
  - *GetUrl* – searches for a service and returns the URL of the provider.
  - *Init* – initializes a coordinator instance.
  - *Merge* – clones state information from existing coordinator by the given URL.
  - *Migrate* – migrates current coordinator to the calling replacement by cloning the state information.
  - *Register* – registers/re-registers a managed component.
  - *Reset* – clears the settings and empties the databases.
  - *Resume* – informs the resume of component identified by the component type and name.
  - *Suspend* – informs the suspension of component identified by the component type and name.

- **ILocalProxy**

  Provides the syntax of required non-web-service functions of a local proxy instance.

  Type: interface

  Functions:

  - *Move* – clones an existing local proxy.
  - *Resume* – resumes the current local proxy.
  - *Start* – starts an instance of local proxy.
  - *Suspend* – suspends the current local proxy.

- **IProxy**

  Provides required web-services functions for a local proxy instance.

Implemented interfaces: *IComponent*

Type: interface

Functions:

- *GetUrl* – searches for a service and gets the URL of the service provider.
- *ResetUrl* – resets the URL of a service given by the service name.
- *GetStatus* – gets the status of current local proxy.
- *Serialize* – serializes current local proxy to the remote calling replacement by returning the state information (i.e. proxy name, service repository and the URL of managing coordinator).

- **IRepository**

  Provides the syntax of required functions of a local proxy repository instance.

  Type: interface

  Functions:

  - *ContainsKey* – determines whether the repository contains a specific key.
  - *ContainsValue* – determines whether the repository contains a specific object.
  - *GetValue* – gets the object by the specified key.
  - *Add* – adds a new object with the specified key.
  - *Remove* – removes an object by the specified key.
  - *Clear* – removes all elements from the repository.

- **KeyElements**

  Stores the state information of a coordinator, which is used to keep the continuity after coordinator migration.

  Type: struct

  Attributes: *Serializable*

  Properties:

  - *Parent* – a *Manager* instance stores the information of the managing coordinator.
  - *Id* – a *String* instance stores the id (name) of current coordinator.
  - *ConsumerIdCounter* – an *Integer* instance uses to help generating unique name for new registered managed consumer.

- *CoordinatorIdCounter* – an *Integer* instance uses to help generating unique name for new registered managed coordinator.
- *Providers* – an *IcomponentDB* instance stores all managed and logical provider proxies.
- *Consumers* – an *IcomponentDB* instance stores all managed and logical consumer proxies.
- *Coordinators* – an *IcomponentDB* instance stores all managed coordinator proxies.

- **LocalProxy**

  Provides functions and properties to manage a local proxy.

  Type: class

  Functions:

  - *Run* – starts a new local proxy or migrates a local proxy from a remote host.
  - *Suspend* – suspends the current local proxy.
  - *Resume* – resumes the current local proxy.

- **Manager**

  Helps the coordinator to store the information of the managing coordinator.

  Type: class

  Atrributes: *Serializable*

  Properties:

  - *Name* – a *String* instance stores the name of the managing coordinator.
  - *Url* – a *String* instance stores the URL of the managing coordinator.
  - *Status* – a *STATUS* instance stores the status of the managing coordinator.
  - *services* – an *ArrayList* instance stores the cross-coordinator services the managing coordinator provided which were required by current coordinator.

- **Provider**

  Represents a remote service provider instance.

  Base Classes: *Component*

  Implemented interfaces: *ICoordinator*

  Type: class

Properties:

- *KeyElements* – a *KeyElements* instance stores all state information, which needs to be cloned to keep continuity when coordinator moves.

- **Proxy**

  Represents a remote provider component.

  Base Classes: *Component*

  Type: class

  Attributes: *Serializable*

  Properties:

  - *Consumers* – an *ArrayList* instance stores the proxies of all ever-called consumers.

- **PseudoConsumer**

  Represents a logical consumer in cross-coordinator service invoking.

  Base Classes: *Consumer*

  Type: class

  Attributes: *Serializable*

- **PseudoProvider**

  Represents a logical provider in cross-coordinator service invoking.

  Base Classes: *Consumer*

  Type: class

  Attributes: *Serializable*

- **ServiceAssist**

  Provides help functions for service provider and help functions to get application settings.

  Type: class

  Properties:

  - *CoordiatorName* – a read-only *String* instance stores the name of published coordinator object.

- *ProxyName* **-** a read-only *String* instance stores the name of published local proxy object.

Functions:

- *GetConfigValue* – static method, gets configured value of an application property specified by the specified name.
- *RegisterEntry* – static method, registers service to remote managing coordinator.
- *Setup* – starts the current service provider by registering all services to the managing coordinator.
- *SetManagerStatus* – static method, resets the status of the managing coordinator of current service provider.
- *ResetManager* – static method, resets the URL of the managing coordinator of current service provider.
- *GetManagerInfo* – static method, gets managing coordinator information.

- **Services**

  Implements web-service functions of a service provider.

  Base Classes: *MarshalByRefObject*

  Implemented interfaces: *IComponent*

  Type: abstract class

  Functions:

  - *Kill* – kills current service provider after migration.

- **SubCoordinator**

  Represents a remote managing coordinator instance.

  Base Classes: *Component*

  Implemented interfaces: *ICoordinator*

  Type: class

  Properties:

  - *services* – an *ArrayList* instance that stores the cross-coordinator services of the coordinator.

  Function:

- *AddServices* – adds a pseudo consumer for logical services the coordinator provided.

B.3 An Example

This example Fibonacci calculation gives a flavor of the P2P-Manifold model. The application includes a service provider, which publishes a Fibonacci web service and a consumer that consumes the service. This example uses standard web-service application architecture and is built by three assemblies/applications:

- **A shared assembly**

  IServices is a shared assembly, which contains interface *IFibonacci* (Figure B-2) and declares the Fibonacci service method. The assembly must be included by both provider and consumer application to unify the method interface. No effort is needed in this assembly to integrate P2P-Manifold model.

```
using System;
namespace IService{
        public interface IFibonacci{
                long Calculate(int num);
        }
}
```

Figure B-2 IService Interface

- **Provider**

  This application (Figure B-3) contains the server-side implementation of the *IFibonacci* interface and publishes the Fibonacci *Calculator* service after startup. The detail settings of service publication are defined in the configuration file "Provider.exe.config" (Figure B-3). In order to integrate P2P-Manifold model, four changes are made:

  - Includes the P2P-Manifold helper library *RegAssist.dll*;
  - Inherits abstract class Services when implementing the service class;
  - Registers the services after startup;
  - Configures the first location of managing coordinator for registration;

```
using System;
using IServices;
using System.Runtime.Remoting;
using Coordination;          //include P2P-Manifold namespace
namespace Provider{
        class myFabonacci:Services, Ifibonacci{      //inherits Services abstract class
                public long Calculate(int num){
                        …
                }
        }
        class Provider{
                 [STAThread]
                static void Main(string[] args){
                        RemotingConfiguration.Configure("Provider.exe.config");
                        ServiceAssist.registerEntry();          //register services to managing coordinator
                        Console.ReadLine();
                }
        }
}
```

Figure B-3 Provider Application

```
<configuration>
        <system.runtime.remoting>
                <application>
                        <channels>
                                <channel ref="http" port="1234" />
                        </channels>
                        <service>
                                <wellknown mode="Singleton"
                                        type="Provider.myFibonacci, Provider"
                                        objectUri="FibonacciService" />
                        </service>
                </application>
        </system.runtime.remoting>

        <appSettings>
                <!—P2P-Manifold setting, coordinator URL -->
                <add key="URL" value="http://localhost:4000"/>
   </appSettings>
</configuration>
```

Figure B-4 Provider Configure File

- **Consumer**

  The consumer application (Figure B-5) contains a sample consumer that consumes the Fibonacci service. It also uses configuration file (Figure B-6) to configure the required custom sink as well as some P2P-Manifold settings.

```
using System;
using System.Runtime.Remoting;
using Coordination;
using IService;
using Coordination; //include P2P-Manifold namespace
public class Client{
        public static void Main(string[] args){
                proxy=new LocalProxy(); //start a local proxy
                RemotingConfiguration.Configure("Client.exe.config");
                string port = ServiceAssist.getConfigValue("PROXY_PORT");
                //always makes call to local proxy
                IFibonacci service =(IFibonacci) Activator.GetObject(typeof(IFibonacci),
                                        "http://localhost:" + port + "/FibonacciService");
                long result = service.Calculate(10);
                Console.WriteLine("Fibonacci(10) = " + result");
        }
}
```

Figure B-5 Consumer Application

Five changes are made to integrate the P2P-Manifold model into the consumer application:

- Include the P2P-Manifold helper library *RegAssist.dll*;
- Make all service calls to local proxy regardless of the provider's location;
- Start a local proxy;
- Configure the local proxy port and the first location of coordinator for first registration;
- Configure the service channel to include the custom sink;

```
<configuration>
  <system.runtime.remoting>
    <application>
                    <channels>
                            <channel ref ="http">
                                    <clientProviders>
                                            <formatter ref="soap" />
                                            <!-- P2P-Manifold setting, custom sink -->
                                            <provider type="Sink.ClientSinkProvider, mySink" />
                                    </clientProviders>
                            </channel>
                    </channels>
     </application>
   </system.runtime.remoting>
   <appSettings>
      <!-- P2P-Manifold setting, coordinator url, mandatory -->
      <add key="COORDINATOR_URL" value="http://localhost:4000"/>
      <!-- P2P-Manifold setting, local proxy bind port  -->
      <add key="PROXY_PORT" value="4040"/>
    </appSettings>
</configuration>
```

Figure B-6 Consumer Configure File

Besides the changes applied to provider and consumer applications, a managing coordinator starts up before running these applications. The coordinator is fully implemented in the .NET P2P-Manifold implementation and is bound to the specified port in the configuration file (Figure B-7).

```
<configuration>
        <system.runtime.remoting>
                <application>
                        <channels>
                                <channel ref="http" port="4000" / >
                        </channels>
                        <service>
                                <wellknown mode="Singleton"
                                        type="Coordination.Coordinator, Coordination"
                                        objectUri="Coordinator" />
                        </service>
                </application>
        </system.runtime.remoting>
</configuration>
```

Figure B-7 Coordinator Configure File