A Plug-and-Play Approach with Distributed Computing Alternatives for Network Configuration Management

by

Syed Kamran Raza

B.E. (Computer Systems),
N.E.D. University of Engineering & Technology, Karachi, Pakistan

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
Faculty of Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario K1S 5B6, Canada

April 19, 1999

The undersigned recommend to the Faculty of Graduate Studies and Research the acceptance of the thesis

A Plug-and-Play Approach with Distributed Computing Alternatives for Network Configuration Management

submitted by

Syed Kamran Raza

in partial fulfillment of the requirements for the degree of

Master of Engineering

Professor Bernard Pagurek
Thesis co-supervisor

Tony White
Thesis co-supervisor

Professor Rafik A. Goubran
Chair, Department of Systems and Computer Engineering

Carleton University
April 1999

Abstract

Network Management is becoming increasingly challenging and difficult with the rapid growth of today's heterogeneous networks. Provisioning of network services is an important configuration management function, which comprises the tedious tasks of installation, initialization, and configuration of the installed service in addition to the tasks of reconfiguration and software distribution on client side. In current management schemes, a network administrator is responsible for performing all these activities *manually* that requires a substantial amount of expertise, effort, time, and cost. One of the possible solutions to this provisioning problem is to *automate* these installation, configuration, and software distribution processes through a plug-and-play approach.

This thesis proposes and analyses a plug-and-play (PnP) approach for configuration management in networks. In this approach, a service provider network component completes the entire configuration and provisioning arrangements itself without human intervention after the component has been plugged into a network. This thesis also demonstrates the applicability of the proposed approach using some state of the art distributed computing technologies, namely CORBA, Mobile Agents, and Jini. An evaluation of the applied technologies is also performed whose results suggest that currently Jini is the most appropriate and ideal technology for supporting plug-and-play environments.

بسم الله الرحمن الرحيم

Acknowledgments

I would like to thank my thesis co-supervisor, Professor Bernard Pagurek, for his guidance, encouragement, and support in this research. His vast knowledge and experience in the areas of Network Management and computing proved very beneficial during the course of the thesis. I am also grateful to other co-supervisor, Tony White, for devoting substantial time and energy in thesis-related discussions and for contributing invaluable ideas. I would like to thank Dr. Andrzej Bieszczad who originally suggested the research topic. Working with him had been a good and rewarding experience.

I am thankful to my parents and family members for their support and understanding during the course of my studies.

My colleagues at Network Management Lab have been very helpful and supportive for which I am thankful to them. At the same time, I would also like to pay my gratitude to all those who have helped me in this research.

This research work was supported by grants from Communications and Information Technology Ontario (CITO) and Natural Sciences & Engineering Research Council (NSERC), Canada.

Table of Contents

CHAPTER 3: THE PLUG-AND-PLAY APPROACH 34

CHAPTER 4: CORBA FOR THE PNP APPROACH 68

List of Figures

Figures in Appendices

Appendix - A

Appendix - B

List of Acronyms

ACL	Access Control List
AEE	Agent Execution Environment
API	Application Programmer Interface
CDS	Cell Directory Service
CMIP	Common Management Information Protocol
COD	Code On Demand
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COSS	Common Object Services Specification
DCE	Distributed Computing Environment
DCE-CIOP	DCE Common Inter-ORB Protocol
DCOM	Distributed Component Object Model
DES	Data Encryption Standard
DHCP	Dynamic Host Control Protocol
DII	Dynamic Invocation Interface
DIP	Dual-Inline Package
DNS	Domain Name Server
DSA	Digital Signature Algorithm
DSI	Dynamic Skeleton Interface
EJB	Enterprise Java Beans
ESIOP	Environment-Specific Inter-ORB Protocol
GDA	Global Directory Agent
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IFR	Interface Repository
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IP	Internet Protocol
IPX	Internetwork Packet Exchange

IRQ	Interrupt Request
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JDK	Java Development Kit
JRMP	Java Remote Method Protocol
JVM	Java Virtual Machine
LAN	Local Area Network
LUS	Lookup Service
MA	Mobile Agent
MAF	Mobile Agent Facility
MASIF	Mobile Agent System Interoperability Facility
MbD	Management by Delegation
MCD	Mobile Code Daemon
MCT	Mobile Code Toolkit
MD-5	Message Digest-5
MIB	Management Information Base
NMS	Network Management System
NVRAM	Non-Volatile Random Access Memory
OA	Object Adapter
ODP	Open Distributed Processing
ODP-RM	ODP Reference Model
OLE	Object Linking and Embedding
OMA	Object Management Architecture
OMG	Object Management Group
ONC-RPC	Open Network Computing - RPC
OR	Object Reference
ORB	Object Request Broker
OSF	Open Software Foundation
OSI	Open Systems Interconnection
PC	Personal Computer
PnP	Plug-and-Play
POA	Portable Object Adapter
REV	Remote Evaluation
RMI	Remote Method Invocation
RP	Remote Programming
RPC	Remote Procedure Call

SHA-1	Secure Hashing Algorithm-1
SIG	Special Interest Group
SNMP	Simple Network Management Protocol
SRD	Service Requirement Discovery
SR-VMC	Service Requirement -VMC
SSL	Secure Socket Layer
TCP	Transport Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modeling Language
VM	Virtual Machine
VMC	Virtual Managed Component

Chapter 1

Introduction

1.1 THESIS OBJECTIVE

The objective of this thesis is to ***propose*** and ***demonstrate*** a plug-and-play approach for installation, configuration, and software distribution of network components using CORBA, Mobile Agents, and Jini distributed technologies.

The plug-and-play (PnP) approach, originally proposed for peripheral management of stand-alone PCs, has been transformed in this thesis for configuration management of network components. For this purpose, a complete set of requirements was developed and a network printer service was taken as an example to identify and analyze the challenges posed during installation, configuration, and software distribution tasks under this approach.

Analysis of the results suggests that, currently, Jini technology is the ideal choice for supporting plug-and-play environments because it fulfils the core requirements of the proposed approach. Although Mobile Agent technology provides flexibility and ease-of-use, it suffers from security and standardization problems. CORBA was found to be a very comprehensive and mature technology but it is complex to use. Furthermore, several important features of CORBA are not yet available in the implemented ORBs.

Finally, the research suggests that standard bodies -- such as Object Management Group -- should define the major requirements and common facilities for Jini and Mobile

Agent frameworks in order to address the issues of standardization and inter-operability.

1.2 MOTIVATION

Management of network components and resources is one of the main concerns for a network manager. This management is becoming increasingly complex and challenging due to the rapid growth and heterogeneous nature of today’s computer and telecommunication networks. Configuration Management, one of the five functional areas of Network Management as defined by the International Organization for Standardization (ISO), is mainly concerned with the initialization, maintenance, reconfiguration and shutting down of network resources and services such as host, servers, routers, printers, etc. In currently available methods, a network administrator is responsible for performing all these activities *manually*, which requires a substantial amount of expertise, effort, time, and cost.

The motivation for this research was to identify the problems in currently deployed configuration schemes and then to propose and implement an approach that would facilitate and automate the installation, configuration, and software distribution tasks for network components. This thesis was also motivated by the fact that various currently available distributed computing technologies seem feasible for network management applications and thus an analysis should be performed to find out that which one may best serve as an enabling technology for demonstrating the applicability of the proposed PnP approach. As a corollary, this thesis finds the proposed PnP approach to be an interesting application for studying and analyzing these distributed technologies for network management.

1.2.1 Problem Description

One of the important tasks in configuration management is the provisioning of network services. This service provisioning applies broadly to two types of services -- software services and hardware services. Software services include applications such as mail servers and domain name servers, while hardware services are concerned with provisioning of network resources such as a printer, disk storage, or a router. Service provisioning includes the tasks of service installation, service activation, and service utilization. Since provisioning of software services and hardware services is not very different from each other, this thesis chooses hardware network components[1] to illustrate the provisioning problem and then to provide its solution.

Components on a typical network can be divided into two broad categories -- Service Providers and Clients. The provisioning of a “service provider component” in a network requires installation and configuration of the component itself and then the advertisement of its service over the network. The process of service provisioning is completed with arranging, distributing, and activating the appropriate service module on all the interested clients. In addition to this, requirement of constantly updating of service modules with service upgrades and changes in existing network configuration pose further challenges during service provisioning. All the aforementioned provisioning tasks require substantial time, effort and the expertise of network administrators. Furthermore, the problem

---

[1] From this point onward, the term *network component* will refer to a network hardware device.

becomes more complex with increasing size and heterogeneity of a network and varies with nature of the service component. For example, installation of a router in a network does not require a significant re-configuration of the existing client components but requires substantial efforts in setting complex and cumbersome parameters on the router (service side). On the other hand, adding a printer to a network requires configuration both at the printer and client sides.

1.2.2 Proposed Approach

One of the possible approaches to tackling the aforementioned problems is to automate the installation, configuration, and service provisioning processes of network components by defining a *plug-and-play* approach [35], [2]. In this approach, when a service provider component is plugged into the network, it completes the entire configuration and provisioning arrangements itself without human intervention. Further, the component configures client network components for its own service utilization. This way, no one else but the added service component takes care of provisioning of services in a plug-and-play manner.

1.3 THESIS CONTRIBUTION

This thesis identifies the problems of service provisioning for Configuration Management and proposes a possible “Plug-and-Play” approach for tackling these problems. The applicability of the proposed approach is demonstrated by using some of the currently available distributed technologies.

The thesis makes a number of contributions as follows:

1. It proposes a generic approach for ***automating*** the installation, configuration, and software distribution for network components. For this purpose, it defines a network service provider component as a plug-and-play (PnP) component. A software engineering methodology has been used for identification and categorization of participant components under this approach to identify their roles and responsibilities. In addition to this, the methodology outlines a plug-and-play scenario by defining the interactions among these components.

2. The contribution also includes the specification of requirements for PnP approach that are, to a large extent, independent of applied technologies.

3. This thesis examines several distributed computing technologies for facilitating network management tasks in general and implementing the proposed generic PnP scheme in particular. The study includes distributed technologies like RMI, CORBA, DCOM, Mobile Agents, JavaSpaces, and Jini.

4. This research tests the applicability of the proposed PnP approach by *designing* and *implementing* it using Mobile Agent, Jini, and CORBA[2] technologies. Furthermore, the use of these technologies provides a better understanding of the value of these technologies for network management applications.

5. The thesis also provides a qualitative (functional level) evaluation of the applied distributed technologies by comparing their pros and cons. Finally, this research

---

[2] design only

provides several recommendations to researcher community and industry.

1.4 THESIS ORGANIZATION

This thesis is organized to present a chronological view of the approaches taken in this research. The research started with the identification and capability evaluation of existing distributed computing technologies, followed by a complete specification and description for the proposed PnP approach. The demonstration of the generic PnP approach started with CORBA technology. Having found a number of problems in the CORBA approach, the work was extended to explore Mobile Agent technology for this particular application. A Mobile Agent based implementation satisfied many requirements of the proposed approach but still suffered from several drawbacks. Recently, Jini technology made its mark on the computing horizon by addressing several shortcomings of the previous applied technologies. Having found substantial support for ‘Plug-and-Play’ environments in Jini, the thesis work concluded with a Jini-based PnP implementation.

This thesis has 6 more chapters.

Chapter 2 starts by providing a brief overview of Network Management and Distributed Computing. Later, a study of current distributed computing technologies like RMI, CORBA, DCOM, Mobile Agent, and Jini is carried out in the chapter.

Chapter 3 presents a complete specification and description of the proposed PnP approach for network components. The specification covers design, implementation requirements, and associated issues in detail.

A design of the CORBA-based PnP approach is presented in chapter 4 along with a short evaluation.

Chapters 5 and 6 provide the implementation, analysis, and evaluation of the proposed PnP approach under Mobile Agent and Jini technologies respectively. These two chapters, along with chapter 3, form the core of this thesis.

The thesis concludes with chapter 7 by summarizing the key thesis messages and providing an insight to researchers and industry. Finally, the chapter also suggests several future research opportunities.

Chapter 2

Network Management and Distributed Computing

This chapter provides the background knowledge that is needed for the development of the ideas presented later in this thesis. First, a brief overview of Network Management, its architecture and functional areas has been provided. This is followed by a description of some of the functions that are required for Configuration Management.

This chapter also provides a study of currently available distributed technologies that seem feasible for performing Network Management tasks in general and Configuration Management in particular. The importance of Java for writing management applications is also discussed during this study. Finally, the technologies that can be used as the enabling technologies for implementing and demonstrating the idea of plug-and-play network components have been discussed.

2.1 NETWORK MANAGEMENT

Network Management addresses primary concerns regarding network control and monitoring. The goal of Network Management is to keep the network operational and reliable by managing network components and resources in an effective manner. Therefore, a Network Management System (NMS) performs the tasks of network configuration, functioning, and maintenance in addition to the management of network faults and data.

2.1.1 ISO Network Management Model

International Organization for Standardization has defined a Network Management model [1] that specifies the most pertinent requirements for managing networks. The model breaks down the requirements into five functional areas that are described as follows.

Fault Management

Fault management is the most widely implemented aspect of Network Management. It is important because faults may occur anytime in networks, causing problems like network shutdown or performance degradation. Fault management's aim is to detect, isolate, log, notify users of, and automatically fix a network problem (fault) up to the possible extent so that the network remains up and running despite faults.

Configuration Management

The hardware and software components are the actual building blocks of a network. Since all such components may have operational problems and flaws that might affect network operation, therefore monitoring of network and system configuration information are essential tasks. Moreover, these components need to be installed, configured, managed and tracked by an NMS. Configuration management addresses all of these requirements.

Security Management

The goal of security management is to protect and control access to network resources

---

[1] as part of its system management specification of Open Systems Interconnection (OSI) reference model

and sensitive information. The functions of a security subsystem include identification of sensitive resources and information, specification of a mapping between such resources and user sets, monitoring of access points, and logging any access to such access points. All this requires the use of authorization mechanisms.

Performance Management

The goal of performance management is to measure, monitor, and make available the various aspects of network performance in terms of variables like traffic, throughput, response time and utilization. The performance management comprises the activities for gathering and analyzing network statistics and trends, as well as determining appropriate thresholds for each important variable.

Accounting Management

Accounting management is about measuring network resource utilization parameters so that an individual or a group use of a network and its resources can be regulated. It requires identification of the costs associated with a network resource and establishment of charges (e.g., billing) for their actual consumption.

2.1.2 Architecture of Network Management System

A classical NMS architecture is based on a typical client-server model that involves two kinds of participants -- network elements that need to be managed and a management application that manages these elements. The managed devices (nodes/network elements) run management software that is called an *agent* and the management application usually resides on a host that is termed a ‘network control host’. The network control host runs a Network Management application that usually provides an operator interface to an

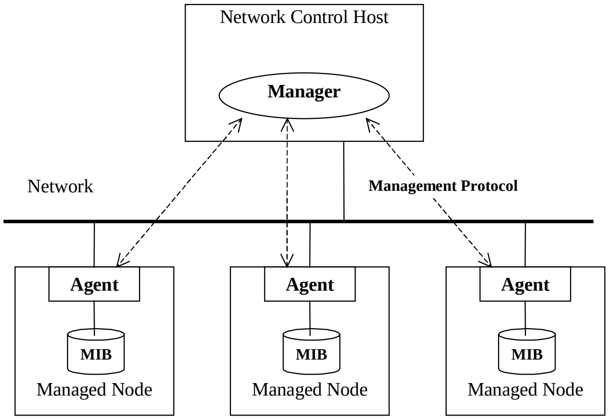


Figure 2.1: Network Management System Architecture

administrator/user for the management purposes. It also runs management software that acts as a *manager*. The agents may send alerts (traps) to the management application under certain circumstances and the manager takes the appropriate actions (such as operator notification, logging, shutdown, etc.) on receipt of the alerts from an agent. The manager not only relies on traps; it may also poll an agent regularly for monitoring purposes. The agent compiles and stores the information regarding the configuration and the behavior (variables) of a managed node in a management database that is called a *Management Information Base* (MIB). A MIB holds the management information that is exchanged between an agent and a manager using some management protocol. There are two major management protocols in practice today -- the Simple Network Management Protocol (SNMP) and the Common Management Information Protocol (CMIP). CMIP comes from an ISO body while the Internet Engineering Task Force (IETF) standardizes SNMP. The ubiquity of the Internet and the simplicity of the protocol are the major factors that have given SNMP an edge over its ISO counterpart. That is why, the SNMP is a more popular and widely deployed protocol. The architecture of a typical Network Management System is shown in Figure 2.1.

2.1.3 Distributed Network Management

Since the architecture described in the previous section comprises a centralized strategy for Network Management, it suffers from the problems of lack of scalability, fault tolerance, flexibility and re-configurability. Furthermore, the current NMSs have data and programs that exist at many sites but their control lack the notion of distribution.

As centralized computing is giving way more and more to distributed computing, Network Management is also becoming distributed. A distributed NMS replaces the single manager model with many mid-level managers that are distributed across the network. The mid-level managers are given limited access for network control and monitoring functions, while a central manager still has the global access and the control rights. Distributed Network Management provides many advantages when compared to centralized approach. These advantages include greater scalability, de-centralization, administrative autonomy, and no-single point of failure.

An important paradigm that has been proposed for Distributed Network Management is the ‘Management By Delegation’ (MbD) [12]. MbD provides distributed, flexible, scalable, partly autonomous, and robust Network Management by distributing intelligence to the networked devices with the help of delegated management applications. The application of mobile agent technology in the area of Network Management has been largely motivated by this concept of MbD.

2.2 CONFIGURATION MANAGEMENT REQUIREMENTS

This section presents the major functions and requirements for Configuration Management area. The focus is to understand how these functions are achieved in the current NMSs. This understanding helps us later in addressing the associated issues with the help of distributed computing technologies.

Following are the main functions that are defined for Configuration Management [36]:

Define configuration information

This function is related to the specification of resources (PCs, printers, routers, bridges, modems, virtual circuits, software services, etc.) and their attributes. The pertinent attributes for resources are name, address, IDs, state, operational characteristics, and software versions. SNMP and CMIP adopt slightly different approaches to fulfil this requirement; i.e., SNMP uses structured lists of data fields, while CMIP takes an object-oriented approach.

Set and modify attribute values

The attributes of a managed entity (resource/service) may be set to a value. For this purpose, a manager remotely sets and modifies the attributes in agents by using the management protocol messages. This operation requires some authorization for the user who is trying to perform this operation.

Define and modify relationships

There is a need to describe an association/connection between network components. It is called a *relationship definition* and its examples include topology, hierarchy, roles, etc. Moreover, configuration management should include provisions for (online) modification of the existing relationships among components and resources. These relationships are defined and modified by a user (network administrator) in current approaches.

Distribute the software

Configuration Management should provide the mechanisms to distribute and upgrade the software of managed entities, where managed entities may include hosts, servers, workstations, printers, routers, gateways and services. In addition to the device software, there may also be a requirement for providing additional device-specific data to devices (e.g., routing table information for a router). Furthermore, the users should also be able to examine, update, and manage different versions of software.

All the aforementioned requirements are the focus of this thesis. The thesis, thus, emphasizes on the components that may auto-install and auto-configure by distributing their software themselves and may provide the mechanisms for *upgrades* without user intervention. A Plug-and-Play approach is proposed for this purpose in chapter 3.

The following sections provide a survey of several available distributed technologies for identifying some enabling technologies[2] that seem appropriate for implementing the proposed PnP approach.

2.3 JAVA FOR NETWORK MANAGEMENT

Java is a programming language developed by Sun Microsystems Inc., which is simple, object-oriented, interpreted, architecture-neutral, portable, multi-threaded and dynamic [40]. It brings together the best concepts and syntax from other existing languages, while overcoming many of their shortcomings.

---

[2] This research regards Java also as a technology.

Java was designed to solve the problem of cross-platform application development for meeting the needs of the developers that work in a heterogeneous network environment. The technique of compiling machine-independent byte codes and the use of a platform-specific Java Virtual Machine provides portability to the applications and allows them to execute securely, independent of the operating system and the hardware architectures. Furthermore, an implementation of the TCP/IP protocol as part of the Java language makes the development of network applications easier.

The power of Java comes from its rich set of libraries, support for networking, facilities for code migration and verification, and features like dynamic class loading and security.

A Network Management application requires two main features in a language -- the support for the heterogeneity of networks and the networking APIs. Java provides both of these features and thus proves to be an ideal language for developing a network management application. That is why, Java is the major implementation language for both the PnP application and the underlying distributed technologies.

2.4 DISTRIBUTED COMPUTING TECHNOLOGIES

2.4.1 Evolution of Computing

Computing has evolved significantly in the last three decades. It started with the *standalone computers* and has kept evolving since then. During its evolution, it has passed through *mainframe computing* to today’s *desktop computing*. At the same time, the software also evolved from monolithic applications to client/server application for enabling *client/server computing*. Today, computers and network technologies are merging to create a new approach to computing, called the *Network Computing*. In the mid to late 90s, the evolution of technologies for distributed objects (e.g., CORBA), component-ware (e.g., COM, DCOM, Beans), programming languages (such as Java) and World Wide Web have had a significant effect on the area of network computing. Recently, with the introduction of powerful distributed technologies like Jini, computing is entering an era of *supercomputing*, thus making Sun Microsystems' dream true -- "The Network Is The Computer" ™.

2.4.2 Distributed Computing

Computing is termed *distributed* when the processing and data are spread out over more than one computer, usually over a network. A distributed computing environment makes a collection of loosely connected systems appear to be a single entity [50]. Such an environment is characterized by the physical separation of computing units, administrative autonomy and heterogeneity and provides many benefits like:

- resource sharing
- performance (due to parallel computing)
- availability (due to de-centralization)
- fault tolerance
- scalability

At the same time, this introduces many major challenges for developers and managers that include issues like:

- naming

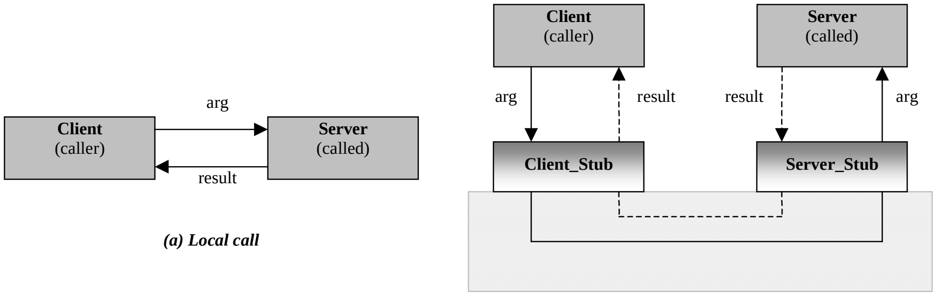
- security
- manageability

A distributed computing technology uses a client/server computing model that is essentially based on the Remote Procedure Call (RPC) mechanism. Several other distributed technologies, however, also use *objects* and *mobile agents* as a means of communication.

The following survey starts with a description of RPC, which is followed by a brief study of several existing distributed technologies. The efforts for open systems (i.e., DCE, ODP) are also discussed for the sake of completeness.

2.4.2.1 RPC: The fundamental technology

The client/server model is one of the most popular forms of distribution. In this model, one entity (a client) makes a request and the other (a server) provides the service. The processing flow is characteristically synchronous (blocking) in nature; i.e., a client initiates an activity and waits until the server has completed the activity. The communication occurs from a client to a server and then from the server back to the client. For the client/server (C/S) programming model, one can use messages as a means of interaction between clients and servers but the tasks related to message passing may be quite tedious and error-prone. Therefore, to ease the task of C/S programming, a remote procedure call mechanism was developed. The idea was to model C/S interactions in a way that would hide the complex details, such as the low-level message passing, from a programmer.



(a) Local call

(b) Remote Procedure Call

Figure 2.2: RPC mechanism

RPC programming is a paradigm that models the C/S interactions as procedure calls and provides a mechanism for invoking the services that reside outside of the execution environment of the caller. Due to its support for distribution, the RPC mechanism serves as the *fundamental technology* for several distributed technologies and frameworks discussed in this thesis. This fact emphasizes the need for a detailed description of the RPC mechanism.

Figure 2.2 shows the mechanism of a remote procedure call and its difference from a local call. The entity that is called a *stub* plays the central role in case of a remote procedure call. A stub is a communication interface, which implements the RPC protocol, and a client uses this stub to talk to a server. To invoke a remote procedure, a client invokes the client-side stub through a local procedure call. The client stub takes the call, marshals[3] the parameters and sends the invocation request to the server side. At the server side, the server stub receives the call, un-marshals the parameters and converts them back

---

[3] Marshalling is the process of converting data into a network transferable form.

to the original form. The server object now executes the required procedure and returns the result to the client. This result also goes through the processes of marshalling and un-marshalling at the server and client side respectively.

A client must, however, know the server interface in order to invoke a call on a remote server. This interface ties the client and server applications together by defining the set of procedures and attributes of a server object using an Interface Definition Language (IDL).

The major differences and issues that distinguish a remote procedure call from a local call are:

- name resolution
- parameter passing
- security
- performance and latency

Due to the distribution, the client and server may exist in different execution spaces or on different machine. Therefore, there is a need for naming the procedures and programs (or object) uniquely so that a client can find them easily. Further, different machines may have different architectures and different internal data representation in a typical distributed environment. The problem, then, is to enable clients and servers to sending data to each other such that each comprehends it. To resolve this issue, RPC uses a standard representation for the data that is to be sent across a network. Since RPC provides a powerful access to client, it is also vulnerable from a security point of view.

Therefore, there is a need for secure RPC that should use authentication and encryption techniques. Finally, the issues like scalability and resilience to failures raise the performance concerns.

The two most prominent RPC systems in use are Sun Microsystems's ONC-RPC (or Sun-RPC) and Open Software Foundation's DCE-RPC.

2.4.2.2 Open Systems

An open system is a vendor-transparent environment where hardware, software, and networks of different forms (from different vendors) can be intermixed. An open system exhibits the following properties:

- Interoperability
- Portability
- Integration

Since the distributed computing and processing environments require these properties, therefore specification of open system standards is essential for these environments.

Open Distributed Processing

Open Distributed Processing (ODP) is a standardization effort for distributed processing and defines a set of standards that are produced by ISO and International Telecommunication Union (ITU-T). The goal of ODP is to provide ease of use to a user in many areas of distributed processing and to hide the nature and means of distribution from an end user.

The ODP provides a uniform and homogeneous view of a system using the *transparencies*. The transparencies are ODP's abstraction mechanism, which masks the technical details from a user. This presents a user with *what* is happening and hides *how* it is happening. These transparencies cover almost all the major requirements and issues of a distributed system such as location, access, migration, transaction, persistence, etc. ODP transparencies are provided by the software elements that are called the ODP *functions*. ODP also provides an effective and disciplined approach for developing distributed systems through the *viewpoint* concept. There are five viewpoints -- Enterprise, Information, Computational, Engineering, and Technology. These viewpoints separate the concerns at different levels and provide a guideline and mechanism for ODP system design. The combination of ODP transparencies, functions, and viewpoints define the ODP *Reference Model* (ODP-RM) in which viewpoints define various functions that provide actual transparencies.

Since the ODP presents reference architecture for distributed processing, therefore, the standards like CORBA and DCOM fit within the ODP framework. These standards, however, implement only some of the facilities that are proposed by the ODP functions [20].

Distributed Computing Environment

Distributed Computing Environment (DCE) is a framework for distributed computing [33] that is defined by Open Software Foundation (OSF). Its primary goal is to overcome the problem that a distributed application faces due to the *heterogeneity* of transport protocols, computer architecture, and operating systems. DCE provides the services that allow a distributed application to interact with a collection of heterogeneous computers, operating systems and networks as if they were a single system. Therefore, DCE is middle-ware that provides a layer between the OS, network and distributed applications as shown in Figure 2.3.

The core component in the DCE architecture is a *cell*. A cell is a grouping of number of machines, users, and resources that may be managed as a single unit. The ability to divide a large environment into independently manageable units facilitates the management of distributed components and makes the DCE architecture highly scalable.

The DCE defines several distributed services and facilities. The RPC facility provides the core mechanism for communication between client and server. The use of Cell Directory Service (CDS) with a Global Directory Agent (GDA) enables the name resolution among clients and servers. The DCE Thread service supports creation, management, and synchronization of threads in a multi-threaded application. Other services such as security, time, and file are also defined in DCE.

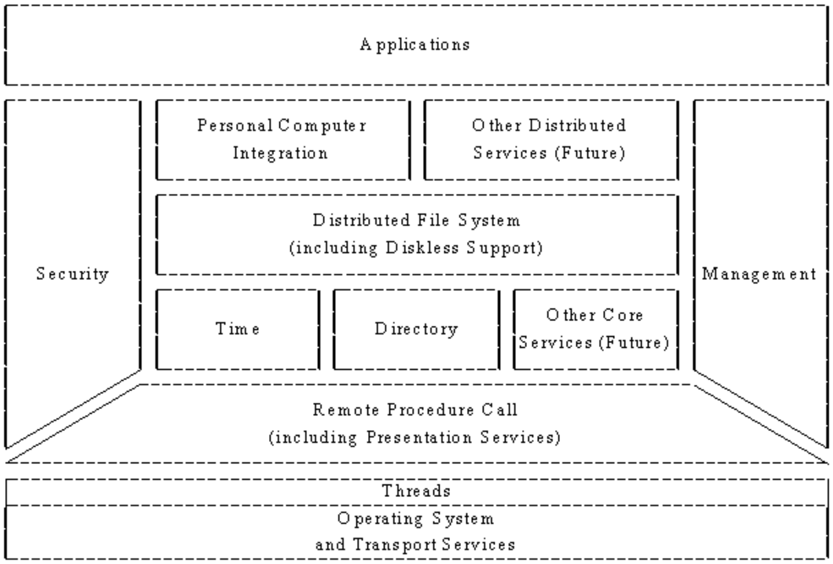


Figure 2.3: Distributed Computing Environment

In short, DCE is middle-ware for distributed applications and addresses the issue of heterogeneity like CORBA. DCE, however, provides a lower-level programming model than CORBA does. Furthermore, DCE is not a fully object-oriented framework.

2.4.3 Distributed Object Technologies

The evolution of object-oriented programming paradigm has its impact on distributed computing, leading to the development of several object-based distributed technologies. These technologies use the basic notion of a RPC but communicate using objects instead of unstructured data. CORBA, DCOM, and RMI are the prominent examples of currently available distributed object technologies. Following is given a brief survey of several existing distributed object technologies[4].

2.4.3.1 CORBA

Object Management Group (OMG) is world's largest software consortium that defines an open-ended distributed environment specification called the Object Management Architecture (OMA). One of the key components of OMA is a distributed object framework, called the Common Object Request Broker Architecture (CORBA) [30]. CORBA provides a middleware component, called the Object Request Broker (ORB), which acts as an object bus over which objects may interact transparently with other local or remote objects. CORBA is not an application or a programming environment; it is a framework that defines an *abstract* object model. Since a detailed

---

[4] This survey mainly discusses Java-based technologies.

description of CORBA is presented in chapter 4, only a high-level view of the technology is presented here.

A CORBA object is presented to the outside world as an interface that has a set of methods. Clients of a CORBA object acquire the object reference of a particular interface and then use this interface to invoke the method on a remote object in such a manner as if the object was located in their own address space. ORB is responsible for all the mechanisms required to find the object's implementation, prepare the object to receive the request, and finally to communicate the request. Similar to the RPC mechanism, the invocation of a method in CORBA requires the use of a stub on the client side and a skeleton on the server side. In addition to the static method invocation through stubs and skeletons, CORBA also provides dynamic invocation mechanisms.

CORBA addresses important issues of heterogeneity and interoperability by defining the Internet Inter-ORB Protocol (IIOP) that allows CORBA compliant ORBs to inter-operate. CORBA also allows integration of the foreign object systems, such as DCOM, by using gateways and bridge protocols.

Another important component of OMA is the Common Object Services Specification (COSS). The COSS [28] specify the services that are used by the objects on an ORB. These services include Trading, Naming, Security, Life Cycle, Persistence, Concurrency and several others.

2.4.3.2 DCOM

The major competitor of CORBA is Microsoft Corporation's Distributed Component Object Model (DCOM). DCOM is a distributed object model [24] that enables components to communicate directly over a network. Like CORBA, DCOM is also a middleware but is based on OSF's DCE-RPC specification.

To understand DCOM, we first need to understand the Component Object Model (COM). Microsoft's COM started as a mechanism for component-based development in the Windows environment. The initial motive was to enable independently developed Windows applications to communicate with each other. Thus, Object Linking and Embedding (OLE) was introduced in the Windows to allow such a communication. Realizing the need for a more general OLE model (that would allow any other binary component in addition to Windows applications), Microsoft developed the COM, which was first introduced in OLE2. COM components, possibly written in different languages, use the standard interfaces and methodologies to pass data on a local machine. With the popularity of distributed computing and Internet, Microsoft immediately extended COM to define DCOM that supports communication across a network. Microsoft calls it a *'COM with a longer wire'* since it uses the same methodology to talk across the network that is used for the inter-process communication among COM components on a local machine. This distributed communication feature is achieved using a *proxy* on a local machine and a *stub* on the remote side. Almost all the Microsoft's current technologies (renamed from OLE to ActiveX) are built around COM/DCOM.

DCOM is similar to CORBA in many respects. It provides the distributed object communication and defines the services like Naming, Security, Trading, etc. The main problem with DCOM is that it is relatively an immature and closed technology (Windows only). The Windows-only feature is also a positive point for DCOM as the user-friendly environment of Windows attracts new programmers, as well as those that are already familiar with the Windows and COM programming.

2.4.3.3 Java RMI

Remote Method Invocation (RMI) is the Java way of implementing an RPC mechanism. RMI is an object-oriented technology that allows seamless computing between a Java client object and a Java server (remote) object. It is the technology that forms the basis for distributed computing in the Java environment ([41], [37]) and thus serves several other distributed frameworks and technologies.

Java RMI defines interfaces and protocols for a remote object that runs on a JVM. The methods of a remote object can be invoked from another object, running on another JVM (potentially on a different host). An invocation appears to travel between a client and server but, in reality, it passes through the RMI stack on both the systems [5]. Figure 2.4 shows the RMI architecture where all the remote objects are represented by their stubs on the client side and by a skeleton on the server side. These stubs and skeleton are responsible for implementing the RPC mechanism.

The RMI mechanism differs from a plain RPC mechanism due to the presence of actual objects. RMI uses the class-loading feature of Java that helps clients and servers for loading required classes at runtime. These classes may include classes for stubs, skeletons, parameter objects, and return objects. Actual method invocation uses the Java Remote Method Protocol (JRMP) which is the RMI wire protocol and is closely related to the CORBA’s IIOP. The JRMP is a simple protocol, which uses sockets as a means of communication and allows method invocation, parameter passing and return-values from

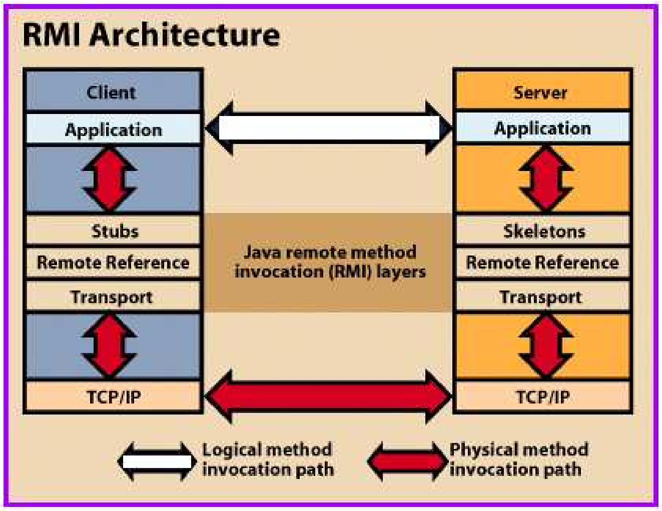


Figure 2.4: RMI Architecture [5]

one JVM to another across the network. The Java Object Serialization facility is used to convert object parameters into their network transferable representations. RMI fulfils the naming requirement by defining a centralized registry that serves as a name repository for the objects and it maps a name to the actual object reference. RMI also supports distributed garbage collection and implements security features that are essential for distributed applications.

Until now, this study has been discussing RMI as a 'Javatized-RPC' for objects. RMI can also be treated as an ORB -- a 'Javatized-CORBA' -- that provides the distributed object facilities for communication, location, etc. RMI is, however, less powerful and limited in many respects when compared to CORBA. For example, RMI is just a Java-only solution, uses its own proprietary JRMP protocol, allows only the static invocations, and defines no common object services except Naming and basic Security service. The other main source of problems with RMI technology is its lack of scalability and heterogeneity. On the other hand, the strength of RMI is its relative ease of use when compared to other technologies. It is also best suited for small and 100% Java-based applications.

In summary, RMI provides a mechanism that may serve as a good infrastructure for building distributed frameworks. This is why several Mobile Agent Frameworks, Enterprise Java Beans (EJB), and Jini are built upon RMI. An understanding of RMI is essential for readers since this thesis also uses Mobile Agent and Jini technologies.

2.4.3.4 Jini

Jini is a recently introduced technology from Sun Microsystems Inc. that simplifies the access and delivery of network services [42], [39]. It is a Java-based technology that is built on top of RMI. The Jini architecture provides mechanisms for building a ***federation*** of devices and applications, where each device or application may offer its resources and services to other members of the federation and may use other resources when needed. In other words, the applications may also dynamically interact without any prior knowledge of each other’s capabilities. For this to happen, Jini defines a service brokering architecture through a lookup service and a set of service protocols that are used by clients and server components to interact. The federation concept is made possible through a set of libraries and certain conventions provided by Jini.

Jini not only provides a networking infrastructure; it also facilitates the development of distributed applications. The technology provides several important distributed facilities like events, leasing, transactions, and security that make the life of a developer much easier. Therefore, network services may easily be developed on top of Jini infrastructure using the Jini’s distributed programming model.

From Jini's point of view, everything is a service, provided that it can be represented as a Java object with particular service interfaces. Examples of services include hardware devices, storage, computation, software, etc. Current release of Jini software provides two ready-made services -- Transaction Manager and JavaSpaces -- that are built on top of Jini infrastructure. JavaSpaces is marketed as a separate distributed technology by Sun Microsystems and hence is discussed in a separate section.

2.4.3.5 JavaSpaces

JavaSpaces is a service that runs on top of the Jini infrastructure and provides a way of building distributed systems. JavaSpaces is also a technology that allows dynamic sharing, communication and co-ordination of Java objects [38]. For this to happen, it provides a single object repository, called the *space*. A space provides a reliable distributed storage system for objects by holding the entries (typed group of objects) that a user may read/write/take (retrieve). The space also provides matching and notification mechanisms. The participants may communicate anonymously via JavaSpaces and the only knowledge they share is the knowledge of the entries they use to communicate with the space. This way, JavaSpaces de-couples service providers and clients.

In short, the JavaSpaces architecture is designed to help solve two related problems -- distributed persistence and design of co-operative protocols [38].

It is worth mentioning here that the entry-based architecture of JavaSpaces is strongly influenced by tuple-space concept of Linda systems [53], which was proposed by Dr. David Gelertner of Yale University.

2.5 MOBILE AGENT TECHNOLOGY

Most of the distributed technologies discussed so far have been RPC-based and use stationary software components (static agents). There is another type of computing paradigm for the distributed systems, which is called the Remote Programming (RP). The RP uses movement of code (procedures) from an originator to a destination computer as opposed to the RPC’s static method calls. The code is thus called a *mobile code*.

A mobile code performs the task remotely that is delegated to it without interacting with its originator. A simple example of the mobile code is an applet. However, an applet does not necessarily keep its state across the network and it also cannot move on its own. The more interesting mobile code is one that may move autonomously from machines to machines, while carrying its state. In other words, a mobile code represents a *program that moves while it is running*. Such a running application that has a migration facility is called a mobile agent. Stated simply, a mobile agent is a software object that may *autonomously* move from one host to another host over the network in order to perform its task [10].

The systems based on Mobile Agent technology promise to provide applications in many domains, such as distributed information retrieval, telecom service management, electronic commerce, real time interactions and network management, to name but a few. The technology has its drawbacks too. Of these drawbacks, lack of standardization and security issues are the major ones. The standardization of this technology are, at last, being pursued by OMG through the efforts like Mobile Agent System Interoperability Facility (MASIF) specification that provides guidelines for interoperability among different agent systems [29].

Mobile Agent technology requires a programming language and an execution environment to run the mobile code. Currently, various execution environments, written in different languages, are available. The examples include IBM’s Aglets [15], ObjectSpace’s Voyager [32], General Magic’s Odyssey [11], Grasshopper from IKV++ [16], and Mobile Code Toolkit ([34],[49]) of Network Management Lab at Carleton University.

2.6. SELECTION AMONG AVAILABLE ALTERNATIVES

The comprehensive study of existing enabling distributed technologies leaves us in a position to select some enabling technologies that seem most appropriate for demonstrating the proposed PnP idea. Of the surveyed technologies, the following three technologies are chosen:

1. CORBA
2. Mobile Agents
3. Jini

2.6.1 CORBA

CORBA is a promising technology that is comprehensive, *mature*, and a result of a significant standardization activity. It is suitable for the development of distributed applications due to platform neutrality, interoperability, and open standard. Furthermore, dynamic invocation support and common object services like Naming, Trading, Security, make CORBA more powerful framework for distributed applications. All these features make CORBA the technology that may enable the proposed plug-and-play approach.

Although DCOM is another candidate that competes with CORBA, it is not chosen mainly because of its closed features. Chapter 4 of this thesis describes a CORBA-based plug-and-play approach.

2.6.2 Mobile Agents

Mobile Agent technology has its advantages in many areas as described earlier, but the advantage that has impressed the author most is the *flexibility* provided by a mobile agent based application. The technology is a push technology so that a piece of code can be injected over the network in an asynchronous manner for performing task on behalf of a user. These features have motivated researchers to explore the use of mobile agent technology for the network management purposes as well. In the scope of this thesis, mobile agents are used for solving the configuration management problem for the network components. The mobile agent based PnP approach is described in chapter 5.

2.6.3 Jini

Sun Microsystems' Jini technology is one of the most promising technologies for designing and implementing the proposed plug-and-play approach. Jini provides the right networking infrastructure that may be used to dynamically add/remove/use a service in a federation. It allows creation of a network of semi-intelligent devices (such as computers, modems, printers, and routers) that can share resources and processing. Using Jini's discovery, join, and lookup facilities, a device may become part of network immediately after it is plugged-in. These important features of the technology and its distributed facilities make Jini a strong candidate for plug-and-play environments. Of all the surveyed technologies, Jini is the ideal technology for demonstrating and implementing the PnP idea for network components. Jini-based design and implementation of PnP network components is presented in chapter 6.

Chapter 3

The Plug-and-Play Approach

This chapter describes the proposed plug-and-play approach, which provides a simple and *generic* solution to the installation, configuration, and provisioning problem of network components. The chapter presents a high-level design, identifies the main participant components, associates the responsibilities with each participant, and describes a simple scenario for the PnP approach. The requirements and issues associated with the approach are also discussed. Before describing the PnP approach for networks, a brief history and overview of well-known PnP approach for PC peripherals is provided.

3.1 PLUG-AND-PLAY HISTORY

The architecture of PC system has evolved significantly since IBM PC/XT was defined in 1981. The rapid development in the computer industry has accounted for faster processors, bigger hard drives, more memory, and many kinds of peripheral devices. These innovations of computer, their hardware, software, and peripherals have made PCs more flexible and powerful but these innovations have also created new challenges for users. For example, anyone trying to configure a PC or its peripherals faces significant problems. Typical examples of configuration tasks are change of mode of a notebook computer from an office to a mobile setting, and addition of a new device such as a CD-ROM, fax modem, or network card to a PC. In order to configure a PC, the user had to dig into the complexity of the IRQs, DIP switches, jumpers, device drivers, and configuration files. According to a Microsoft's support person, nearly half the calls to the Windows help lines were from the users that were struggling to install or to configure hardware and software [13]. This statement clearly reflects the measure of the difficulty that a PC user had at this time.

To resolve these kinds of issues, the need for a supporting framework was soon realized -- a framework that should enable a home user to buy an accessory, take it home, let it install itself and get it working the very first time. This realization caused Microsoft and Intel to propose a Plug-and-Play framework in March 1993 at the Windows Hardware Engineering conference. Later, two companies founded a Plug-and-Play Association and released its specification in May 1994. This Plug-and-Play had three immediate goals:

- make PCs easier to setup and configure;
- ease the task of installing new software and hardware;
- have features to change configuration of the PCs on the fly and inform the corresponding software applications accordingly.

Microsoft's Windows '95 was the first operating system that had the support for plug-and-play (PnP) feature. This feature was welcomed immediately by PC's manufacturers and peripheral vendors and has gained more and more acceptance and standardization since then.

3.2 MICROSOFT'S PLUG-AND-PLAY FRAMEWORK ARCHITECTURE

A PnP system for a PC is made up of a PnP-OS, a PnP-BIOS, and PnP-hardware devices. The devices identify themselves and declare their resource requirements, whereas, the BIOS accepts and responds to these devices. The PnP-OS is the central component that controls the entire process of loading and configuring device drivers and responds to the hardware changes without user intervention.

Microsoft defines the following functional aspects of a PnP system [26] that are to be managed by a PnP operating system:

- Identification of installed devices.
- Determination of device driver requirement.
- Resolving resource conflicts.
- Loading device drivers.
- Notification of configuration changes.

The Microsoft Windows PnP architecture defines many elements -- Configuration Manager, Bus Enumerators, Resource Arbitrator, and Hardware Tree – that collaborate to achieve the aforementioned PnP capability as shown in next section with the help of an example.

3.2.1 Functionality

Consider a PC that is running a Windows 95 operating system and a user that is using Microsoft Word on that PC for word processing. The user would like to print the document but there is no printer installed with the computer. Now let us assume that a PnP printer is plugged into the PC parallel port. The corresponding *Bus Enumerator* detects the addition of a new printer immediately and determines its resource requirements (such as I/O, IRQ) and passes this information to the *Configuration Manager*. The Configuration Manager compares the information with its *registry* and concludes that a new device has been added. Therefore, it asks the *Resource Arbitrator* to resolve any conflicts. The arbitrator, in turn, rebuilds the existing *hardware tree*. Furthermore, Windows operating system keeps a large repository of device drivers from various vendors for installing the corresponding driver during PnP scenario. If corresponding driver is not found in the repository, the user is prompted to install the driver from a vendor-supplied disk. After the installation of a driver, the registry is updated and the PnP peripheral is up and running at this moment.

The PnP-OS also notifies all the applications, whether running or passive, about any configuration change. Microsoft Word, for example, gets the notification and re-adjusts the application to make the new printer as its default print device. Note that the change in the configuration (i.e., addition of a new printer) is detected on the fly. Even though the application did not know about any printer at launch time, the application was able to detect the presence of a new printer through the PnP mechanism.

3.3 PNP: FROM PC TO NETWORK

It has been identified in the introductory chapter of this thesis that the installation and configuration of a new component on a network constitutes an important network management task. This task, however, is not easy and may require significant time, effort, cost, and expertise from a network administrator. The identified problem regarding configuration management of network components is similar to the problem of installing a PC peripheral in many respects. As the installation and configuration of a PC peripheral was made easy through the introduction of a plug-and-play approach, the configuration management problem of this thesis can also be facilitated by adopting a plug-and-play approach for network components. The idea is to define a PnP system for networks where components can *automate* the cumbersome steps of their installation, configuration, and provisioning.

Therefore, this thesis transplants the plug-and-play idea from the (stand-alone) PC world to the networked world by proposing a generic plug-and-play approach for network components. After some manual steps that are required for their installation in the network, the PnP components make their installation and configuration arrangements themselves without user intervention. It is to be noted that this thesis uses the term “PnP” to refer exclusively to a network component that provides a network service.

3.3.1 Objective

Following are the major goals of the proposed PnP approach for networks:

- Minimize the stress on a network administrator with regards to the component’s configuration issues.
- Make the setup and configuration of network components/services easier.
- Continuos supervision of network for supporting the dynamic nature of the network.

3.3.2 Network-Specific Constraints

The PnP concept of the PC world needs several adjustments when applied in networks. The reason is that there are certain constraints that are only specific to networks and are not found in PC world. Some of these constraints are:

- Interaction between a component and a network is relatively complex.
- Keeping a registry for all the network components and their interaction schemes is almost impossible.
- Provisioning a network component is usually more complicated than installing a simple PC driver.
- No centralized OS is available in networks that could handle the auto-installation scenario.

The rest of this chapter focuses on defining a generic PnP approach, which is implemented in later chapters using three distributed computing technologies. A typical software engineering methodology is applied for synthesizing and specifying the approach. First, a requirement analysis is performed which is followed by a functional and operational description of the proposed approach. Several diagrams that apply the Unified Modeling Language (UML) notation [31] are used to illustrate the design and analysis of the approach.

3.4 REQUIREMENT ANALYSIS

This section starts the analysis part of the proposed approach. Starting with a high-level and an abstract view of the PnP scheme, the section progressively follows the software engineering methodology. In this methodology, participant components are discovered, their roles are established, and finally the responsibilities are assigned.

3.4.1 Identifying the System and Actors

The first step in the analysis is to acquire a good understanding of the *system* and its *environment*. In the context of this thesis, the system of concern is the application that provides a plug-and-play capability and its environment includes the *actors* that use or affect this system via interfaces (e.g., physical network topology, network cards, console etc). Five major actors are discovered that interact with the PnP application as shown in Figure 3.1:

1. Service provider network component (Plug-and-Playable)
2. Client network component
3. Vendor
4. Network Administrator
5. Broker

As stated earlier, a service provider component is a network component that offers service(s) to other network components, whereas, a component that requires or uses an offered service is called a client component. A vendor is the actor that manufactures and supplies a service provider component and its associated software. The broker is an important actor since it serves as glue between clients and server components; i.e., components discover each other using this broker. Finally, a network administrator is the

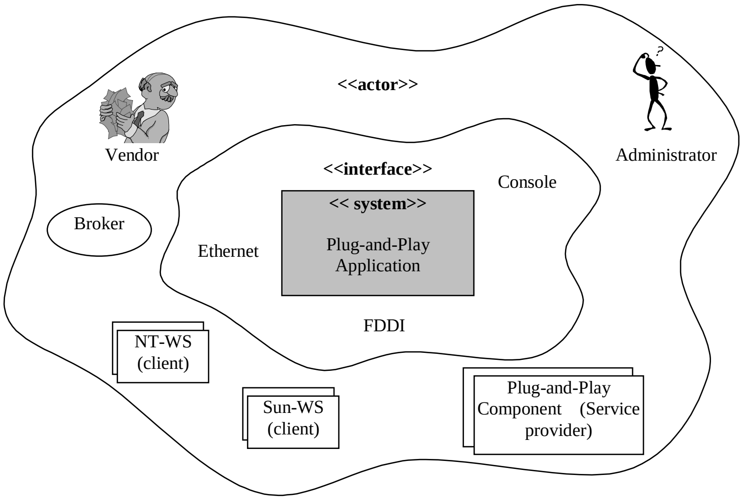


Figure 3.1: System boundary and actors

person who is in charge of managing the network.

3.4.2 Capturing the System Requirements

Once the system boundary and the actors have been identified, the next step in the analysis methodology is to determine the system requirements. Following are the main features that are required of a PnP system:

1. Auto-installation of client/server components.
2. Network discovery.
3. Service provision (PnP component).
4. Service discovery and its utilization (client components).
5. Service upgrade mechanisms.
6. Dynamic configuration.

Despite the best efforts for automating the entire installation and configuration scenario, there are some tasks that cannot be automated and require user intervention. Examples of such tasks include:

- Establishment of a physical connection.
- Arrangement of initialization information for configuration, setup, and security.
- Monitoring of logs and system messages.

3.4.3 Responsibility of each Actor

The aforementioned requirements for a PnP system can be categorized by *roles*. This means that each identified actor must be assigned its associated functionality and requirement. Following is given a list of requirements and responsibilities assigned to the actors of this PnP system.

Service Provider PnP Component

1. Physical installation.
2. Locate and join the network.
3. Requirement of initialization parameters for initial setup and security controls.
4. Registration with its vendor and server-side service module downloading.
5. Service advertisement over the network through a broker.
6. Accept and register client requests.
7. Security and access control for client registration.
8. Arrangement of client-side service modules for its clients.
9. Service upgrading.
10. Notifications to all clients and its vendor on its service withdrawal.

Client Component

1. Physical installation.
2. Locate and join the network.
3. Requirement of initialization parameters for initial setup and security controls.
4. Service requirement.
5. Find and lookup (discover) an appropriate service provider through a broker.
6. Register with the service provider component.
7. Receive and install client-side service module.
8. Utilize the required service.
9. Receive and handle notifications on client-side module upgrades.
10. Security and access control.

11. Remain synchronized with the status change of its service provider components.

Vendor

1. Supply a service provider PnP component with the pre-installed server-side service module and PnP software.
2. Hard-code the vendor-related information in PnP components at manufacture time.
3. Maintain a repository of the service modules (both client and server side).
4. Accept registrations from its service provider PnP components on their startup.
5. Send notifications to corresponding components on service module upgrades.
6. Security and accounting controls on PnP-components registrations and service module downloading.
7. Third party provision (optional).

Broker

1. Facilitate service advertisement of service provider PnP components.
2. Accept requests from client components for locating other components or services.
3. Bind a service and client component.

Network Administrator

1. Setup of initial configuration parameters, security control, and physical hardware installation.
2. Monitor overall system behavior, response, logging etc.

3.4.4 Requirement Modeling

After assignments of responsibilities, the context diagram and the use case diagrams can be used to further illustrate the captured system requirements. A context diagram shows how various actors interact with the main system. The context diagram for the proposed Plug-and-Play system, shown in Figure 3.2, illustrates the main interactions among the identified components and the main application.

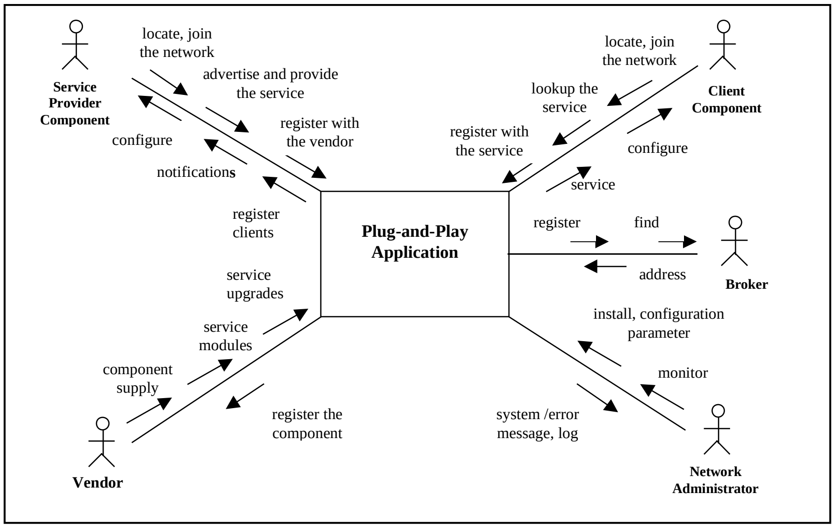


Figure 3.2: PnP system's Context Diagram

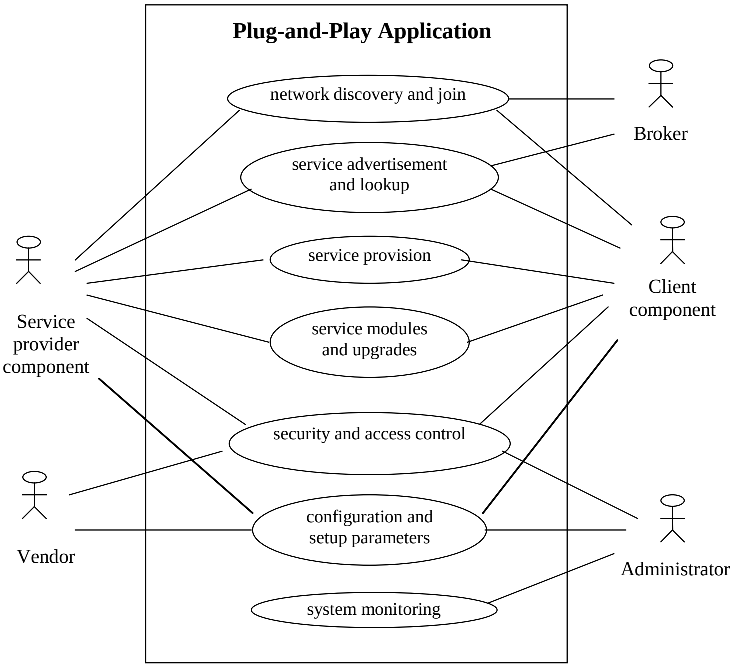


Figure 3.3: PnP system's Use Cases

Another software engineering tool for illustrating the captured system requirements is Use cases. The use cases of an analyzed system emphasize the main functionality of that system. Figure 3.3 represents the use cases of the PnP system where each ellipse corresponds to a use case of the PnP application. It is to be noted that a single use case, in fact, may be derived out of many requirements listed in the earlier section.

3.5 OPERATIONAL AND FUNCTIONAL DESCRIPTION

While previous sections have attempted to present a high-level view of the system, this section analyzes the system more closely by describing its functional and operational characteristics. A generic plug-and-play scenario is presented here that is followed by discussions on several associated issues.

3.5.1 Simple Scenario

Following is a chronological listing of typical events that occur during the installation of a PnP network component. The list describes a simple scenario without discussing the complex issues (such as security, failures and recovery, etc.) and only illustrates the collaborative operations performed by the participant components. Although the scheme incorporates several steps where a task can be achieved in more than one way, this list only enumerates the approach that is most widely deployed in the forthcoming implementations.

1. A vendor ships a PnP component in the market with a pre-installed software and hard-coded information. The vendor also maintains a repository of service modules corresponding to its manufactured components.
2. A network administrator, who is in need of the component, purchases it.

3. The administrator first supplies the initial configuration parameters to the component and then installs the component on the network.
4. The PnP component initializes itself and installation proceeds only if all the self-tests are passed.
5. The PnP component, having attained the communication capability, uses the hard-coded information to locate its vendor and eventually registers with the vendor.
6. The PnP component initially downloads the latest server-side service module from the vendor and activates it.
7. The service offered by the component is advertised over the network through a broker.
8. An interested client finds and locates its desired service using a lookup mechanism (in some schemes, the service may locate its clients).
9. The client shows its interest in the service and thus registers with the corresponding PnP-component.
10. The service provider PnP-component determines the requirement of all the registered clients and arranges the client-side service module from the vendor’s module repository.
11. The service modules are distributed, either by the vendor or by the PnP-component, to the corresponding client components.
12. The modules are installed and activated on the client components.
13. A client may now use the offered service of the PnP-component. With this, the initial service provisioning routine completes.
14. The vendor keeps its repository of service modules upgraded and notifies all the registered PnP-component (and the clients, in some cases) of any subsequent upgrade.
15. Upon receiving an upgrade notification, the PnP-component starts an upgrade routine that repeats the steps for distribution and installation of client-side modules. In case of a server-side upgrade, the PnP-component obtains the latest module and performs a

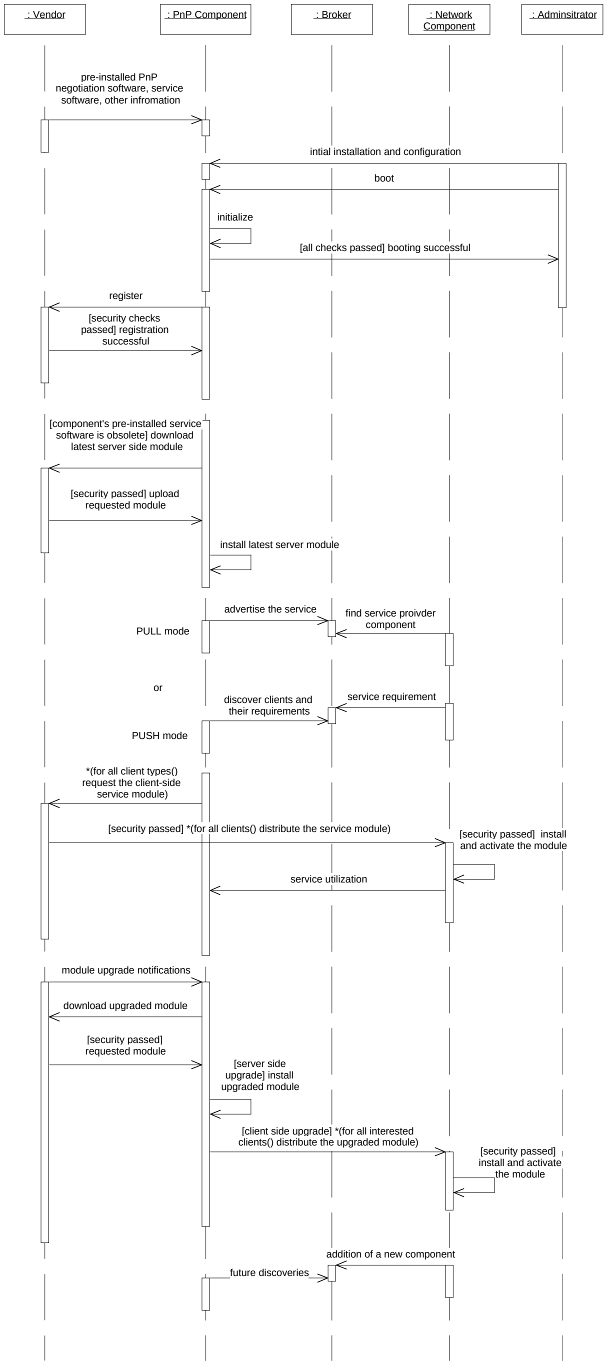


Figure 3.4: Interaction diagram of PnP approach

hot software swapping. In case of a client-side module upgrade, the PnP component arranges delivery of the module to the affected clients.

16. The PnP-component also monitors the supported network in order to detect any new client or a change in the network configuration etc.

17. All the sensitive interactions among the participants require authentication and authorization mechanisms.

Figure 3.4 is a UML interaction diagram that illustrates the described scenario. It is to be noted that the shown scenario does not involve much interactions between an administrator and the network, thus the scheme achieves the objective of minimizing, to a large extent, the human intervention.

3.5.2 Requirements and Issues

Following subsections describe the facilities that are required by this PnP approach for achieving the designated functionality. The issues that are associated with the components and their interactions are also addressed.

3.5.2.1 PnP component software requirements

Analogous to PnP PC peripherals that require a pre-installed PnP negotiation protocol, a service provider PnP-component requires some pre-installed software and information as discussed below.

Pre-installed server-side service module

There is a need for a server-side service module on a PnP component to process clients' requests. This module can be supplied as the component's firmware (as is the usual practice nowadays). The provision of the pre-installed service software on the component simplifies some of the configuration tasks but introduces another problem; i.e., the service module becomes obsolete on the arrival of an upgraded module on its vendor side. Some alternative approaches are discussed later for solving this problem.

Pre-installed PnP negotiation software

A server component needs a pre-installed PnP software in order to be PnP-compliant. One of the related question is that how the vendors would be convinced to support a PnP software in their devices because it will require extra cost, extra time, and extra work on the vendor side. The answer is simple -- by providing a smarter, intelligent device that is plug-and-playable, the vendor may gain an edge over its competitors because the users/customers would prefer to buy a smarter device that could facilitate its provisioning process. At the same time, fear of loss of market share will also force other vendors to adhere to this approach.

Hard-coded information

There is also a need for assigning some initial setup and configuration information to each component. The network administrator provides some of this information (such as customer id, component name, IP address, operation modes etc.) at the time of component's installation. In case of a PnP-component, there is another source of information that comes from the vendor itself. This source is the hard-coded information on the component and it relates largely to the product and the vendor itself (such as the component type, service type, product id, serial number, and model). The information also includes the address of the vendor site so that the component can communicate with the vendor over the network.

3.5.2.2 PnP component hardware requirements

The aforementioned software requirements give rise to certain hardware requirements for the components. Furthermore, the capability of a PnP-component for initiating and controlling the installation process requires more resources on a component. The cumulative hardware requirements for a PnP-component include:

1. firmware
2. significant memory
3. processing capability
4. configuration panel: for passing configuration parameters at installation time

These hardware requirements raise some doubt on the applicability of this PnP scheme on small devices but it is believed that such hardware devices would be available in near future with the rapid advancement of technologies.

3.5.2.3 Initial installation and configuration of a network component

Following are the typical arrangements that a network administrator has to make for the initial installation and configuration of a network component (i.e., both the PnP and client components):

1. Manual hardware connection.
2. Data entry of the initial configuration parameters using component's configuration panel.
3. Allocation and assignment of an IP address to the component.

The assignment of an IP address to the component can be achieved in two ways:

1. Manually: Assigned by the administrator that finds an unused IP address, passes this information to the component using its panel, and finally creates a Domain Name Server (DNS) entry for this new IP address.

2. DHCP: The use of the Dynamic Host Control Protocol (DHCP) provides a mechanism for obtaining an IP address dynamically [18]. That is, a component

serving as a DHCP client can obtain an initial IP address (along with other initial configuration parameters) from a DHCP server over the network. In this case, the component gets the IP address information itself, whereas the task of updating the associated DNS entry is delegated either to the DHCP server or to the DHCP client [19].

Once this initial setup is completed, the component is able to communicate over the network and thus may take full charge of the entire installation and setup process.

3.5.2.4 Registration of a PnP-component with its vendor

Although a vendor may have shipped many PnP components to market, not all of them are installed on a network. To locate the installed components, a registration of a newly installed component with its vendor is required. This registration also helps vendors in notifying their registered PnP components of any future upgrades. The registration information includes PnP component's name, serial number, model, product ID, customer ID, and service type.

During its initialization routine, a PnP-component first tries to contact its vendor for registration. The vendor, on the other hand, maintains a list of all the currently registered PnP-components and uses appropriate security policies to allow registration to the authenticated and authorized components only. After registering, a PnP component may also choose to upgrade its pre-installed module by downloading the *latest* server-side module from the vendor.

3.5.2.5 Dependency on the Internet

From the described scenario, it is clear that plug-and-play components are heavily dependent on an Internet connection for communicating with their vendors. This limitation raises several concerns -- what happens if the component's network is not connected to the Internet? Or how does the component behave if there are *firewalls* to the Internet?

The pre-installed service module at PnP component helps the component for providing the service in the absence of an Internet connection. Furthermore, it is suggested that a soft copy of the service software should also be provided with the component that would enable an administrator to install the software manually in worst cases. Following are the various alternatives for the server-side module arrangement:

1. Use of a pre-installed service module.
2. Manual installation of the service module.
3. Use of auto-installation feature to download the latest module.

The PnP application selects the *auto-installation* option by default.

3.5.2.6 Service Provisioning

Federation of services

One of the goals of the PnP approach is to establish a federation of services where service provider components may advertise their services and clients may locate and access the service through a broker component. This approach suggests the use of a *lookup server* that actually serves as the broker component for the PnP participant components. Since all the components first need to know the location of such a server, therefore, the address of the lookup server is to be provided initially to all the network components at their installation time. Furthermore, a lookup server stores the crucial and important information that should not be lost if the server crashes, therefore, a persistent server is suggested.

Service Lookup using Hierarchical Naming

A service is an entity that is offered by a service provider component and accessed by a client in the network. The common examples are print, scan, storage and routing services. A client broadcasts its requirement over the service network by specifying the required service type and a service provider component replies only if its offered service matches the requested service. This raises a need for defining a common structure for a *service offer* by a PnP-Component and a *service lookup* by a client.

Therefore, a service structure, which is comprised of following two entities, is proposed:

1. service type: basic facility
2. service attributes: other relevant information

A careful distinction needs to be made between a service type and a service attribute. For example, if a client requires *‘color laser postscript printing’* at the speed of 12 pages/minute, then the service type is postscript color printing while its associated attributes are the type and the speed of the printer, i.e.,

service = ‘pnp . printer . color . postscript’ , (laser, 12 pages/minutes, …)

For a service matching purposes, both the service *type* and the service *attributes* are to be matched:

1. Type matching: a service type would match if:
    - The requested service type *exactly* matches the offered service type, or
    - The requested service type constitutes a subset (sub-type) of the offered service type.
2. Attribute matching: all the requested service attributes must be present in the offered service attributes.

It is to be noted that when many services match, it is up to a client to choose the most appropriate service. A hierarchical naming structure for services is suggested for enabling this kind of matching, where the services are derived from a root context and represent some basic service, such as printing. In addition to its attributes and the basic type, a service may also have its sub-types.

The matching mechanism can be best understood with the help of Figure 3.5.

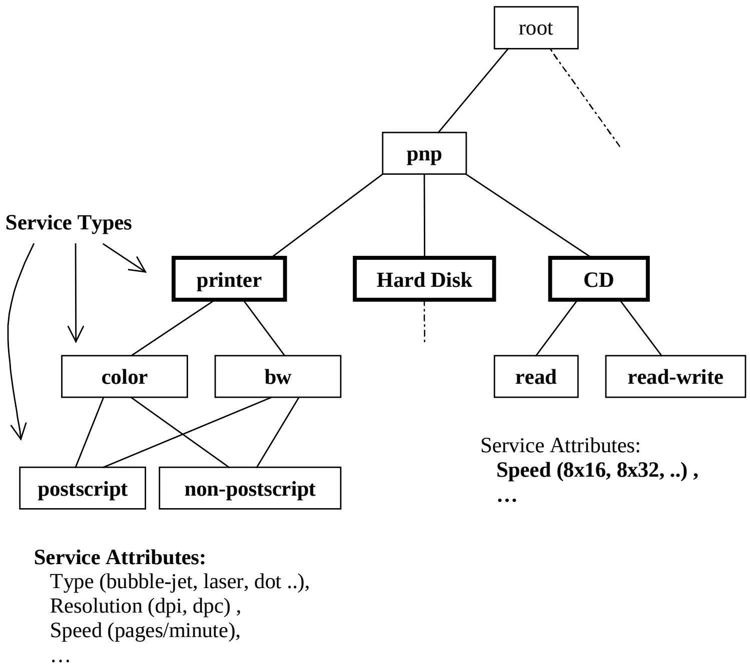


Figure 3.5: Hierarchical naming structure for services

Consider a client that wants to use a *color printer* with the *laser* printing capability. The client’s requirement is

required service = ‘pnp . printer . color’ , (laser)

Note that this client does not care whether printer supports postscript printing or not. Now assume that there are three printers over the network that offer various kinds of print services:

offered service 1 = ’pnp . printer . bw . postscript’ , (laser, 12 pages/minute, …);
offered service 2 = ’pnp . printer . color . postscript’ , (laser, …);
offered service 3 = ’pnp . printer . color . non-postscript’, (dot matrix, …).

According to this PnP specification, the service 2 matches the service requirement completely since it is subtype of the required service and all the requested attributes are also present in this service. The service 3 matches the service type but it does not match the attributes, whereas service 1 does not match the service type in first place. Therefore, the matching fails for both the service 1 and service 3.

It goes without saying that there is a requirement for standardizing the service names, attributes, and hierarchy through a standard body.

Service Advertisement

A service provider PnP component advertises its service after its initialization. However, a place is needed where the component may publish its service. Therefore, the address of the centralized broker (i.e., the lookup server) is initially provided to the component that helps in joining the federation of services over the network. Thus, the component first locates the lookup server and then advertises its service through this server.

Service Requirement

After the service advertisement by a PnP-component, the service is ready for use. Two design alternatives for initiating the service requirement phase are possible:

1. Pull mode: clients raise service requirements and find some service provider component.
2. Push mode: the service provider component searches for any clients that may require its service.

In Pull mode, a service provider (PnP) component advertises its service and then just waits for its clients and it is the responsibility of clients to raise their requirements and find an appropriate service through a lookup server. After finding its required service, the client may communicate with the corresponding service provider component. Note that, a client is required to pass sufficient information to the PnP component in this scheme.

In Push mode, the service provider PnP component initiates the requirement phase and tries to find any clients that are interested in its service. The client components are located through a lookup server but the PnP component needs to know client's requirements in order to determine if a component is a potential client or not. The requirements of each component are to be specified by an administrator and can be placed in a lookup server or on the client machine. A central database may also be used to keep requirements of the clients. Having discovered an interesting client, the PnP component adds the client in its database of the interested clients. The requirement of maintaining this database incurs an overhead on the PnP component side.

Service module Distribution and Activation

Once the service provider has the knowledge of its potential clients and their requirements, it may arrange client-side service modules. Therefore, the PnP component conveys the information related to required service modules to its vendor and the vendor returns the requested service modules from its repository.

The distribution of the modules to their respective clients can be performed in two ways:

1. Distribute the modules directly from the vendor to the clients, or
2. Send the modules from the vendor to the PnP component that eventually distributes it to the clients.

In the first case, the vendor requires sufficient information about the clients (e.g., their name and location) for distributing the module. The vendor also needs to maintain the list of clients, which should actually be the concern of the PnP component. Furthermore, this approach introduces problems like *coupling* between a vendor and network clients, and more processing on the vendor side. The overhead, however, is more significant in the second case since it requires significant memory at the PnP component side for downloading the client-side service modules. The first approach for module distribution is preferred during forthcoming PnP implementations.

After the distribution of the modules to the clients, the next step is the installation and the activation of these modules so that the clients may start utilizing the service. The installation and the activation phase can also be performed in various ways:

1. The modules are self-extracting and auto installing.

2. The client component installs and activates the modules for immediate service use.
3. The modules are *saved* at the client side and installed later by a user.

With this activation phase, the basic PnP configuration scenario is completed and the client components are ready to use the service offered by the corresponding PnP component.

Service Upgrade and Hot-Swapping

Software usually goes through many revisions during its lifetime and thus its updated versions are released from time to time. There are many reasons for upgrading software -- to fix bugs in an earlier release, to add a new feature, to enhance the performance, to support a new technology, etc. Therefore, the installation of the service modules on a client is not sufficient and clients must be kept updated with service upgrade.

The proposed PnP approach makes a vendor responsible for notifying about any upgrade in its repository of service modules. These upgrade notifications are to be sent to all the concerned PnP components that are currently registered with the vendor. Since there is no guarantee of delivery associated with a notification, therefore, an additional polling mechanism is also suggested; i.e., a PnP component should also poll its vendor infrequently for checking the status change or upgrade information.

On receiving an upgrade notification, the PnP component initiates the upgrade routine. The execution of the upgrade routine, however, depends on the type of service upgrade. There are two types of service upgrades:

1. client-side module

2. server-side module

Upgrading a client-side module is simpler as compared to a server-side module upgrade. In this case, the PnP component repeats almost the same procedure that is used for the initial installation of the modules. On the other hand, only a service provider (PnP) component needs to be upgraded if the upgrades are related to a server-side module. The server-side software upgrade is, however, more interesting as it requires an online upgrade of the server without service interruption. Our research group is also performing research in this area to define a safe, reliable, and robust software upgrade mechanism, called the 'Software Hot-Swapping Technique" [1].

This PnP scheme also suggests that upgrades should be provided only to the interested clients. This means that some clients may not want to be upgraded because of various reasons. For example, a stable client may not want to be disturbed for a while after investing a substantial time and energy in attaining the stability. Therefore, some components may opt for '*no-upgrade*' option and thus the PnP component should not interfere with such devices and should upgrade only those clients that have chosen so.

3.5.2.7 Dynamic Configuration

A typical network is dynamic in nature -- it keeps changing as new components are added, old components are removed, the configuration of a component is changed, and so on. Therefore, the responsibility of a PnP component does not end after configuring all the interested network components at the time of its installation. There is an on-going need for keeping track of the ever-changing network conditions in order to configure/reconfigure any new clients. This means that if a new component (potential client) is added to the network, a PnP component should immediately know about its presence and should initiate a service provisioning routine, if required. Similarly, a change in the configuration of an existing and configured client component may require a PnP component to reconfigure the client for service provisioning. The requirement for dynamic configuration is one of the hardest requirements to achieve in this approach and different implementations take different approaches to fulfil this need.

3.5.3 Component Interfaces

This PnP approach has so far presented the components and their interactions. The participant components, however, need to know each other's interfaces in order to collaborate. This approach, therefore, proposes some generic interfaces that must be implemented by the participants for achieving the PnP functionality. The proposed PnP interfaces are:

Vendor interface

A vendor must implement this interface. This interface defines the operations for PnP components' registration, de-registration, and service module interactions.

PnP interface

This interface has to be implemented by a PnP component (i.e., a service provider). The main operations supported by this interface enable clients to register/de-register, and helps in receiving upgrade notifications from the vendor.

Client interface

A client component that uses a service must implement this interface. The typical operations are much the same as the operations provided by a PnP interface and the only exclusion being the registration and de-registration operations.

Admin interface

The management tasks are performed through an admin component that represents a network administrator. This component implements the admin interface that allows PnP participants to communicate and to pass events and error messages to an administrator.

It is to be noted that a component may implement more than one interface. For example, a service provider PnP component may also function as a client component for other services.

3.5.4 Other Important Features

Having described the scenarios of proposed approach, this section now enumerates some important features and facilities that are required for the proposed PnP approach.

3.5.4.1 Errors and recovery

Although the scenarios described until now have assumed no errors and exceptions, this approach has to take into account the failure points of the system and must implement the backtracking and recovery procedures. Since all the possible error conditions and events cannot be enumerated, therefore, only the main failure points that require special consideration are cited here.

Communication breakdown

While two participants are interacting to perform some operation, such as down loading or installing a service module, a communication failure may abort the whole process. There may be various causes for such a failure -- the network may go down, a participant host may die/reboot, or configuration changes during the operation etc.

Therefore, a timeout-and-retry mechanism is to be deployed to detect the conditions when one of the participants (network, originating component, or destination component) dies or crashes during a network interaction. The error is detected by a participant component as it times out for the response from another component. The component tries again and gives up after repeating the process certain number of times. It also sends an error message to the administrator application.

Installed module does not work

Consider an installation or an upgrade of a service module on a client component. Let us say that the service module installation phase succeeds without any problem. However, during its activation phase, it is discovered that the new service module does not function properly, thus rendering the component unusable. For handling such problems, this PnP approach proposes making a backup of the original modules before over-writing them with the upgraded version. The backup is saved at the client side and may be restored if the activation of newly installed module fails.

Vendor out of business

A vendor is an important actor in the PnP approach because it supplies up-to-date service modules to other components. What happens if a vendor dies or goes out of the business? A provision of a third party is also suggested that can provide the required service module in such circumstances. It is, however, the responsibility of a vendor to notify the clients (PnP-components) about such third parties before abandoning its business.

What if the PnP feature fails

Our experience with the plug-and-play peripheral of PCs reminds us that although the PnP architectures provide substantial ease and convenience due to their auto-installation feature, they may cause significant problems if their PnP feature fails. Consider buying a PnP peripheral device and then plugging it into the PC. The PnP feature works well in most cases, however, sometimes the protocol/device fails that leads the user into a deadlock situation where he finds no *manual* way of configuring the component. The solution, then, is either to keep trying with the existing software and hardware or to replace the PnP peripheral with a new peripheral.

This problem has also been taken into the consideration in this PnP approach by suggesting two operating modes for PnP components:

1. Plug-and-Play (PnP) mode: auto-installing
2. Manual mode: can be used when the PnP modes fails

Furthermore, the approach makes an administrator in charge of setting the operating mode of a PnP-component through the configuration panel. By default, the operating mode of a PnP component is set to the *PnP* mode.

Recovery

When failures occur, the PnP system should have capability to recover gracefully. The recovery process, however, depends on the gravity of the failure; i.e., the system may recover itself in some cases while the administrator's intervention may be required in others. It is suggested that all the error and exception events be properly logged and sent to an administrator's console, thus enabling the administrator to take immediate corrective measures if a need arises.

3.5.4.2 Security and Access Control

Distributed systems are more vulnerable to security breaches when compared to traditional systems because there are more places where the system can be attacked. Therefore, a security service is one of the basic requirements for a distributed system.

The main security requirements -- confidentiality, integrity, and accountability -- are achieved in this PnP approach by proposing the following functionality:

1. Identification and authentication of participants
2. Authorization and access control
3. Security administration by an administrator
4. Security control during a communication
5. Security auditing associated with the participants

The approach also recommends the use of security mechanisms that are based on the concept of a *Principal* and *Access Control*.

Identification, authentication and authorization

The identification and authentication together constitute the confidentiality requirement of a system. Therefore, each component is to be assigned a unique ID for meeting the identification requirement. In this PnP approach, the authentication is required whenever a component is registering with other component or a component is requesting a resource or service from other component. The examples include:

- registration of a PnP component with its vendor
- registration of a client’s component with a service provider (PnP) component

- service module downloading by PnP or client components

Since the authorization concerns the integrity of data, therefore, its need arises whenever a component tries to access/modify information on another component. For example, a read from and a write onto a client component by a PnP component should be allowed under authorized access only.

Security Administration

The administrator plays an important role in the security control of the system by managing the security information. This security information is usually supplied during the initial configuration and initialization phase. Typical security management tasks are assigning IDs to the principals (components), creating principal-based ACLs on the client component, etc.

Security during communication

Since the actual communication among participants involves a network, therefore, there are many possible security threats such as disclosure, eavesdropping, masquerading. In addition to other security controls, there is a need to establish some kind of *trust* among the participants in addition to other security controls. Traditional data encryption schemes, such as Data Encryption Standard (DES), may be used to achieve this goal.

Security Auditing

Security auditing is concerned with making the users/principals accountable for their security-related actions and operations. This PnP approach suggests the use of appropriate mechanisms (e.g., logging) for fulfilling this requirement.

3.5.4.3 Logging facility

Accounting is an issue that is closely related to the auditing issue since it primarily concerns with keeping track of the activities and usage of network resources by its users. Therefore, this PnP approach proposes a facility for logging the major activities and events so that an administrator can keep track of the system status, activities, and events. There are several activities in this PnP approach that are log-able. Some examples are:

- initialization of a component
- installation and upgrading of a module
- errors, exceptions and unsuccessful operations
- access and security controls

Furthermore, the logging facility of this PnP approach may also incorporate the delivery of important errors/events to an administrator via a console, an email, or even through some interaction with an intelligent monitoring system, such as an expert system.

3.5.4.4 Trading

Although this PnP approach talks about a service provisioning which is free-of-charge, the fact is that the services offered by an entity are usually charged according to their use. A vendor, for example, may not allow a PnP component to download the service module without paying for it. Similarly, a third party may require charges to be paid before making the shipment of service module. All this means that a PnP-component has to *trade* with such service providers and product manufacturers on behalf of an administrator, thus requiring a trading service. Although the support for a trading facility is already provided in some of the enabling technologies (such as CORBA), it is assumed that the services are provided free-of-charge.

3.5.4.5 Persistence

A distributed system usually comprises several objects that store important information and one does not want to lose this information even if the server crashes or dies, thus requiring the server to be *persistent*. The persistence facility helps in retaining and managing the persistent state of an object and the easiest way to achieve a persistence facility is to save the state (or data) of the system in a file. Since the lookup server of this PnP scheme has to maintain a crucial information about the registered services and components, therefore, it is required to be persistent. Further, this PnP approach proposes use of persistence for keeping a copy of other critical data.

3.6 CONCRETE PNP APPROACHES

This chapter has presented a high level PnP approach for network configuration. The concrete examples of this generic PnP approach are presented next in this thesis by designing the approach in CORBA, Mobile Agents, and Jini distributed technologies. Furthermore, implementations are also performed for the last two mentioned technologies. Although the following concrete approaches conform to the ideas and approach presented in this chapter, the constraints of a particular technology may enforce the use of an alternative mechanism for performing tasks in some cases.

Chapter 4

CORBA for the PnP Approach

This chapter presents a CORBA-based design of plug-and-play approach for network components. The chapter starts with an explanation of Object Management Architecture (OMA) that is followed by an overview of CORBA and its architecture. After CORBA sections, the CORBA-based PnP approach is described in detail. The chapter concludes with an evaluation of the advantages and disadvantages of the suggested approach.

4.1 CORBA OVERVIEW

4.1.1 Why CORBA

It is becoming difficult to get computer and hardware to work together with the increasing complexity and heterogeneity of modern networks. Moreover, different systems use different languages, which also makes software inter-working very difficult. The solution to these problems requires a distributed framework that would satisfy the requirements of distribution, scalability, heterogeneity, and reliability. In recognition of the problems associated with the distributed environments, the Object Management Group (OMG) was formed in 1989. Application integration as its main goal, the OMG has defined open-ended distributed environment specifications, known as Object Management Architecture, which addresses the major issues of multi-vendor portability and inter operability. One of the key components of OMA is the Common Object Request Broker Architecture (CORBA) [30]. CORBA provides a great deal of flexibility in distribution and implementation of today's networked applications through the transparencies related to the programming languages, platforms, operating systems, and network protocols. The key feature of CORBA is its cross-platform communication architecture, which enables definition, transportation, implementation, and invocation of objects in virtually any programming language. All these features make CORBA a powerful enabling technology for network management purposes.

4.1.2 Object Management Architecture

The Object Management Architecture provides a common language for applications and enables the *interoperability at the application level* by defining standard services and interfaces. OMA gives us the big picture of the architecture whose core element is CORBA. Figure 4.1 illustrates the OMA and shows the object interfaces that can be categorized into four main categories:

1. **Object services:** These are standardization of basic services that almost every object needs. The examples of these services include Naming, Trading, Security, Event, Life

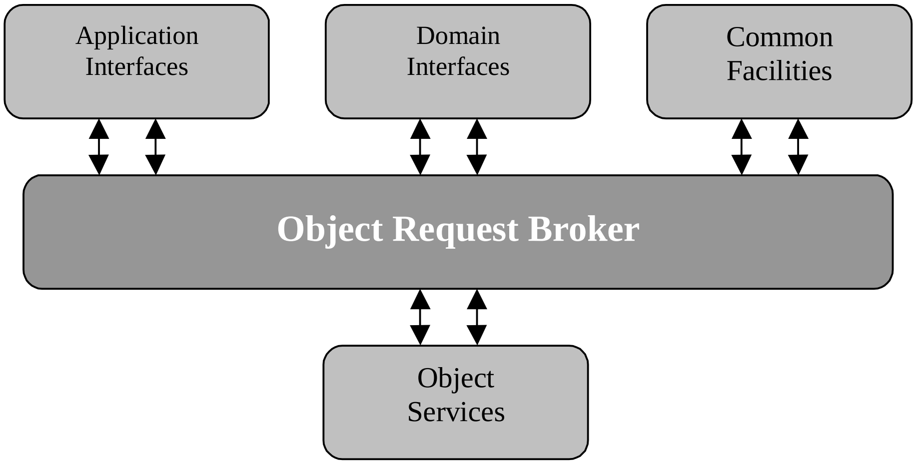


Figure 4.1: Object Management Architecture

Cycle, Concurrency, Transaction, and several others.

2. **Common facilities:** These are the horizontal facilities that provide an intermediate level services to applications. Compound Document management facility is one of the examples of these facilities.
3. **Domain interfaces:** Many industry groups can benefit from standards of their own by defining domain interfaces (vertical facilities) in collaboration with OMG's Special Interest Group (SIG). These domain interfaces provide the standard interfaces that are oriented towards specific application domains, such as health care, financial services, and telecommunication.
4. **Application interfaces:** They are developed specifically for a given application, such as the interfaces of this PnP application. These interfaces are defined by the end-user and not by the OMG.

The Object Request Broker (ORB) of Figure 4.1 is a key component in the architecture of OMA and is explained in the following description of CORBA.

4.1.3 CORBA

Figure 4.2 illustrates the primary components of the CORBA architecture. The high-level view of the architecture yields three main components -- a client, an object implementation and an ORB. A *client* is the entity that wishes to perform an operation on an object, while an *object implementation* is the code and data that actually implements the object by providing the semantics (state and behavior) of the object. The *ORB* is the middleware component that acts as a broker between a client and an object. It is

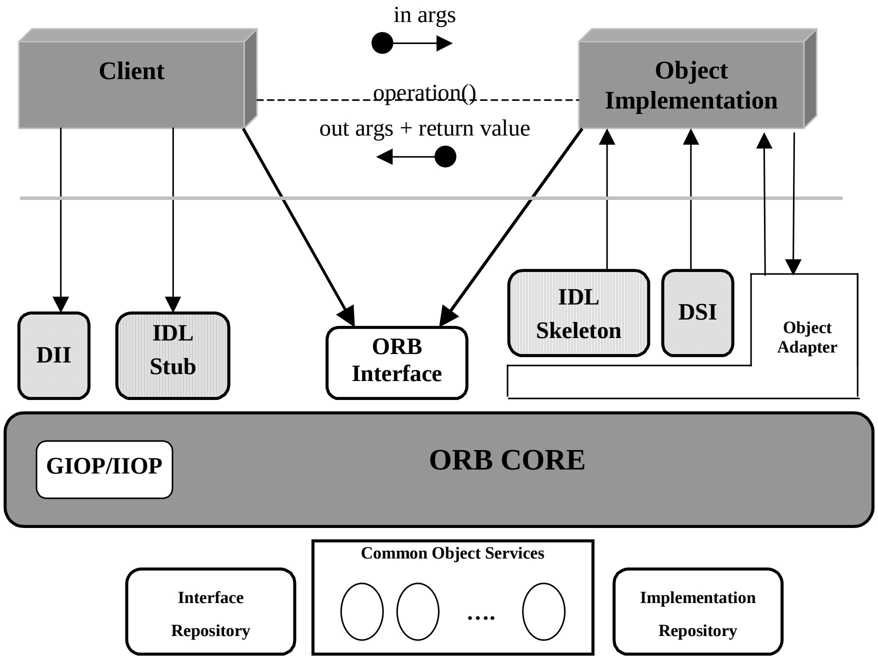


Figure 4.2: CORBA Architecture

responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The ORB functionality is provided either through a daemon process or through library routines that are linked into an executable module along with the client and object implementations.

The services provided by an object are expressed in the form of a contract that serves as an *interface* between an object and the rest of the system. The client knows only the logical structure of the object according to its interface and experiences its behavior through invocations. To the client, the interface represents a promise, while to the object, the interface represents an obligation. The interface consists of operations and their parameters for the object and is defined using the Interface Definition Language (IDL).

The interface[1] definition serves two purposes:

1. inform the potential clients about the services of the object;
2. let the communication infrastructure (i.e., ORB) know the format of all the messages that the object may send or receive, thus allowing the infrastructure to translate data formats among sender and receiver, if required.

IDL provides a declarative, language-independent syntax that allows objects from different programming languages to communicate with each other. There are language mappings, defined by OMG, which determine how the IDL interfaces are mapped to the facilities of a given programming language. These mapping include all major programming languages -- C, C++, Smalltalk, Java, COBOL, and Ada.

Each object has a unique reference that a client may obtain by accessing the CORBA directory services (Naming or Trader). This reference is then used to invoke the actual operation on that object. CORBA defines two modes for the invocation of request:

1. static invocation
2. dynamic invocation

4.1.3.1 Static invocation

Static invocation uses statically defined IDL information about the operations and their associated parameters for an object. The IDL information is compiled to generate

---

[1] In this chapter, the word ‘interface’ always refers to an IDL definition of an operation.

two parts -- a *stub* and a *skeleton*. The stub part is used by a client and provides access to the IDL defined operations supported by an object implementation, whereas the skeleton part is a server side counter part for the stub that delivers requests to CORBA object implementations. The client creates and invokes the service request over the ORB using object's stub. The stub works directly with the client's ORB to marshal parameters associated with the request, whereas the server's ORB and the skeleton co-operate to un-marshall these parameters on the target object side.

4.1.3.2 Dynamic invocation

Some applications, for which priori knowledge of IDL (stub/skeleton) is not possible, require more flexible approach. For example, any application that deals with dynamic environments of a network may require a dynamic approach to interact with other applications. Therefore, a mechanism is required that would allow applications to use a runtime *dynamic typing*[2] instead of the static stub definitions. Moreover, it is desirable to discover and use the new objects as soon as they are added to any ORB. CORBA provides a dynamic invocation mechanism through two interfaces -- Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI). Since this PnP application requires dynamic configurations, the dynamic invocation is an important feature for this application.

---

[2] Note that both static and dynamic approaches use dynamic bindings

Dynamic Invocation Interface

The DII mechanism allows a client to invoke a request on any object without having the compile-time (static) knowledge of the object interface. This helps a client in invoking the dynamic requests on the objects:

- for which the client has no stub;
- that are newly added to the network;
- that are discovered through a directory service (Naming/Trader).

DII is actually a CORBA interface, defined in OMG-IDL, which allows a client to construct an invocation request dynamically. For this mechanism to work, CORBA requires an ORB to store the IDL definition for all of its objects in a repository, called the *Interface Repository* (IFR). The IFR is itself a CORBA object with standard interfaces that are used by an object implementation and clients to store and retrieve the IDL definitions (operation and parameters). The CORBA requires each ORB to have access to at least one IFR.

Dynamic Skeleton Interface

The DSI mechanism is a server-side counterpart of the DII and allows servers to be written without having statically linked skeletons. A static approach requires a "*separate*" skeleton for each interface of the object, whereas the DSI offers a "*single*" standard interface that may be used for serving many interfaces of an actual object. In other words, DSI is opposite to DII. In DII, the client knows that it was using DII but the object implementation could not tell, whereas in DSI, the object implementation knows whether the request is coming from a static skeleton or DSI but the client does not know.

The DSI is also a crucial and important feature for networks as it allows any local object implementation to connect dynamically to an ORB and helps in ORB-to-ORB communication (through bridging).

4.1.3.3 Other ORB Elements

Every CORBA object requires an object reference (OR) which is interpreted by the ORB during the dispatch of a request to the corresponding object. An OR can also be portable across an ORB boundary and is thus called an Interoperable Object Reference (IOR). Another important server-side ORB element is an *Implementation Repository*, which is used by an ORB to *locate* and *activate* the implementation of objects. Therefore, an implementation repository is responsible for maintaining a registry of all the known servers, recording the server host and port information, and activating the servers on demand.

Another important ORB element is the *ORB Interface*, which is used for initialization purposes both on the client and server sides. At start-up, clients and servers need a handle for the ORB and the object references for services like Naming, Trading, Interface Repository and Implementation Repository. All of this initial information is made available through the help of this ORB interface.

Since different object implementations may incorporate a variety of mechanisms that require different working processes, it is logical that an object implementation should not interact directly with an ORB but should use some kind of adapter. CORBA specifies an *Object Adapter* (OA) for this purpose that serves as the glue between an ORB and the object implementations, thus allowing any ORB to operate with the full range of object types. The responsibilities of an OA include registration of an implementation, generation and interpretation of object references, mapping of an OR to the object implementation, activating and deactivating implementation, and invocation of the methods on the object. Current CORBA specification describe a Portable Object Adapter (POA), which allows applications and their programming language object implementations to be portable between ORBs supplied by different vendors.

4.1.4 Interoperability and ORB

When communicating among objects that reside on different ORBs, there is a need for an inter-ORB communication protocol to fulfil the heterogeneity needs. There are two ways of putting this heterogeneity into perspective:

1. get all the ORBs to speak the same protocol;
2. use bridges to translate protocols when ORBs use different protocols.

GIOP

For ORB-to-ORB interactions, OMG has defined General Inter-ORB Protocol (GIOP) that is an abstract, simple, and easy-to-implement connection-oriented protocol. Due to the ubiquity of the Internet, the Internet Inter-ORB Protocol (IIOP) has been developed for TCP/IP

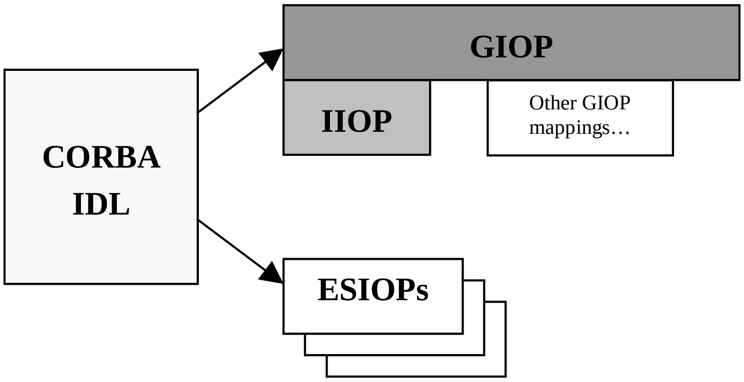


Figure 4.3: Inter-ORB Protocol Relationships

as a concrete representation of the abstract GIOP to support the Internet world. Like IIOP, the mappings of GIOP are also possible for other protocols, such as IPX. Interoperability issues can also be addressed by defining an open-ended set of Environment-Specific IOP (ESIOP). One such example is the DCE Common IOP (DCE-CIOP).

Figure 4.3 shows various Inter-ORB protocols and their relationship. This PnP application assumes the use of IIOP for addressing interoperability among various ORBs.

4.2 CORBA BASED PLUG-AND-PLAY APPROACH

The generic plug-and-play approach presented in chapter 3 can be implemented using CORBA. The approach is realized as a distributed application, where the participant components reside as CORBA objects on an ORB. Following is the description of some of the major concepts in the CORBA-based approach:

1. Every participant entity and network component in this PnP scheme is a CORBA object that has a unique object reference.
2. The components define standard interfaces for plug-and-play negotiations.
3. The components communicate with each other through interface invocation for achieving the PnP functionality.
4. The ORB acts as the underlying communication structure and middleware component, which takes care of all this communication by providing the appropriate transparencies.
5. Several CORBA services (Naming, Security, Events, etc.) are used [28] to provide the basic object services to this distributed application.

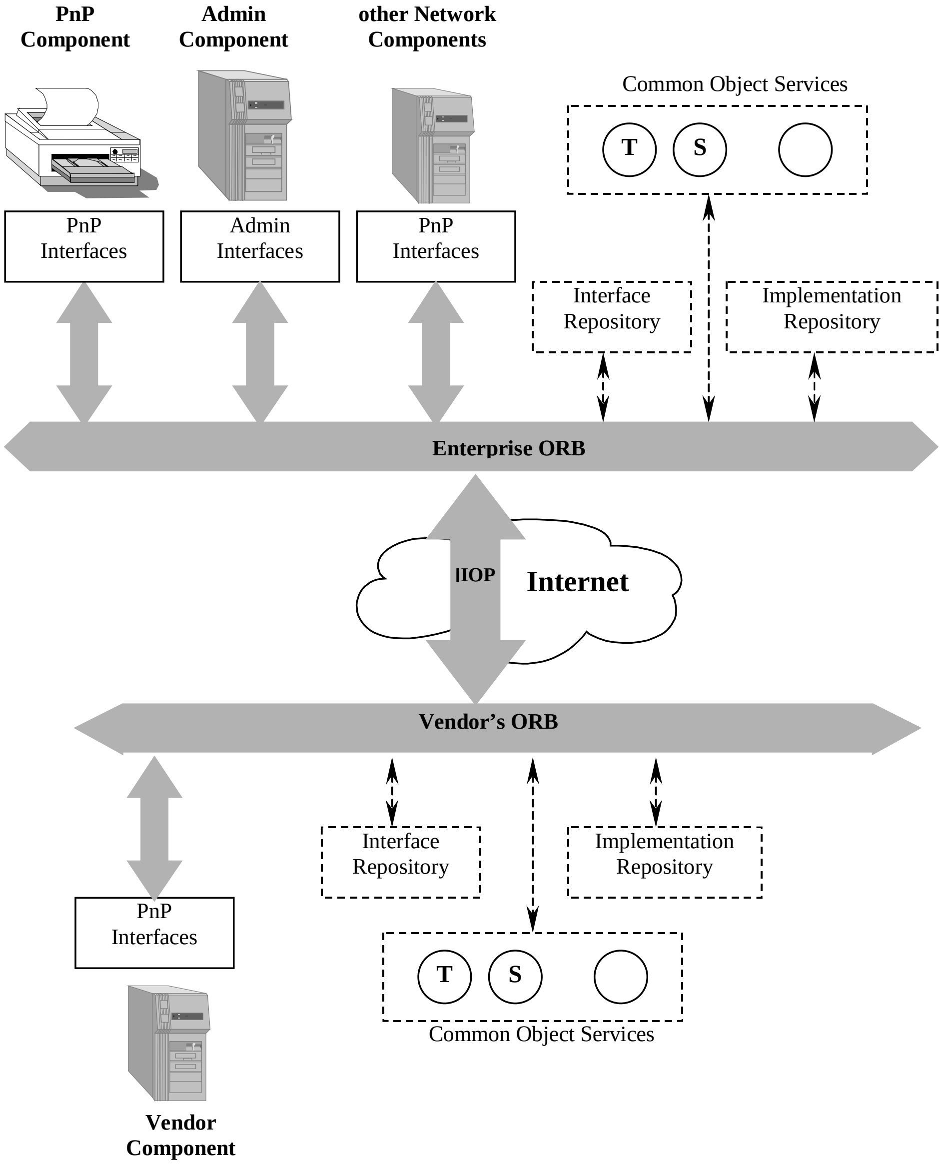


Figure 4.4: CORBA based PnP approach

Figure 4.4 illustrates the deployment of the CORBA-based PnP approach. The diagram does not show all the building blocks of an ORB -- such as stubs, skeletons, ORB interface, and POA -- for the sake of simplicity.

4.2.1 Main Building Blocks

Before describing the PnP scheme, this section describes the important and major building blocks of the PnP scheme.

Domains and ORBs

The PnP application is divided into two domains -- the vendor domain that comprises the vendor site and the vendor component, and the enterprise-domain that involves the network of interest, running various clients and server (PnP) network components. Each of these domains has an ORB on which the respective components reside. These ORBs not only bind local components but also inter-operate for enabling remote bindings. Although the ORBs may be different from a technology perspective (network, protocol, and platform), it is assumed that both the ORBs use the same protocol and communicate using the IIOP. Given the existence of a specification for bridges, above assumption does not pose a significant limitation.

Participants as CORBA Objects

Of the five identified actors (section 3.4.1), the PnP, client, vendor, and admin component are represented as CORBA objects, whereas the underlying ORB and its directory service play the role of the broker actor. Every hardware component (PnP and other network components) needs to have appropriate ORB functionality pre-installed on it in order to function as a CORBA object. Furthermore, every CORBA object in this system has a unique OR, assigned by its host ORB, which remains valid until the object is explicitly deleted. The participating components may be implemented in a variety of ways using different programming languages but they define the standard PnP interfaces proposed earlier in section 3.5.3.

In CORBA’s method invocation, a caller is termed as a *client* and the called is termed as a *server*. Note that CORBA allows an object to play the role of a client and server simultaneously. For objects to communicate and collaborate in this manner, the client object needs to know:

1. server component's interface and its semantics
2. server component's reference

The publication of well-defined interfaces by each server component solves the first requirement, whereas, the second problem is addressed using an object reference (IOR).

Interfaces

For objects to 'plug-and-play' together in a useful way, clients must know exactly what they can expect from other objects. Therefore, the interface definitions are used to define the required contract/agreement among the clients and server objects. The OMG-IDL can be used to write the PnP interfaces (section 3.5.3) in CORBA. An example PnP interface, written in IDL, is shown below. The shown interface is implemented by a

```
module PnP {

    struct ComponentInfo {
       string name;
       string owner;
       string address;
       ….
    };

    interface vendor {

       readonly attribute vendorName;
       ……

       boolean register(in ComponentInfo pnpc) raises SecurityException;
       void deregister(in long long registrationID);

    };
};
```

vendor component and helps the PnP components in registering and de-registering with their vendors.

At this moment, the PnP interfaces are proprietary to this PnP approach and thus fall under the category of ‘application interfaces’ on the OMA picture (Fig. 4.1). However, It is assumed that these CORBA-based PnP interfaces will be standardized through some SIG in future and the vendors will start manufacturing the components with the built-in support for these interfaces.

Interface and Implementation Repositories

Since all CORBA based invocations (static or dynamic) require IDL interfaces, it is proposed that both the vendor’s and enterprise’s ORB maintain an interface repository, which is accessible to other ORBs as well. The IFR at the enterprise ORB stores the interface definitions of network client components, PnP components, and admin component, whereas the vendor’s IFR maintains the interface definitions for the vendor component.

An Implementation Repository is also required on both the described domains for storing and retrieving the object implementations. This repository also provides other features such as the activation and deactivation of objects and their servers.

4.2.2 Useful CORBA Services

This PnP application benefits heavily from CORBA’s object services. Although CORBA specifications describe 16 common services [28] that are used (directly or indirectly) by most of the CORBA applications, this section highlights only those services that provide a key functionality to this PnP application.

Naming and Trading service

The clients of an ORB-based system require a directory service for locating the objects across the network. CORBA provides two directory services -- Naming and Trading. The Naming service is the principal mechanism that provides a mapping from a name to an object reference and is used when clients know whom they want to call. The Naming service, however, does not fulfil all the requirements in dynamic environments. For instance, a client component looking for a printer service that may print color and double sided at a speed of 10 pages/min requires a sophisticated directory service. In this case, the client only knows what it wants but does not know which object best fits the requirement, therefore, it needs to search by category. To fulfil such requirements, CORBA provides a Trading service. The trading service not only serves the basic function of name resolution, it also allows server objects to register their service offers and clients to find the required services. In other words, if a Naming service works as *White pages*, a Trading service performs the functions of *Yellow Pages* in an ORB.

Since CORBA’s trading service fits well in the role of a broker component (section 3.4.1), it is suggested that each ORB in this PnP application run a trader service. The components first obtain the OR of the trading service through the ORB-interface. Using this OR, a PnP component may advertise its service while the client components may discover (find) the services they require.

Security service

CORBA provides a detailed specification for a Security service, specifying mechanisms that provide end-to-end security in multi-vendor object networks. It also specifies the functional aspects for security service as discussed earlier in section 3.5.4.2. The CORBA security service defines appropriate security interfaces for the authentication, access control, recording audit events, and security administration. All these features satisfy the security requirements of this PnP application.

Event service

The proposed PnP approach of chapter 3 comprised several scenarios where the components required the event and notification services for communicating asynchronously. The CORBA framework provides an Event and Notification service that may be useful for this purpose. This event service is designed to provide the 'change notification'. This means that a change in an object generates an event that is propagated to all interested parties. The objects that are changeable and generate events are called the *supplier* objects, while other objects that are interested in the notification of the event are called the *consumers* of the event. The suppliers and consumers of an event communicate using another CORBA object, called the e*vent channel*.

Property service

The participant components of the PnP application require definition of some properties at run time. For example, after initial installation of a network component, the administrator may want to configure the component by supplying the initial configuration parameters (e.g., name, location, owner, and id). This requirement is fulfilled by CORBA's Property service that allows associating the properties (i.e., <name,value> tuples) dynamically with an object using the standard interface calls.

4.2.3 Dynamic Invocations

Although the static method invocation works well for most of the PnP negotiations, an alternative dynamic approach may be chosen in some cases. For example, the clients that have no knowledge of the interface of an object can use DII, whereas DSI can be used when we want our object implementation to be changed and upgraded without disturbing or affecting its clients. The use of the dynamic invocation feature provides us with many advantages that are listed in the last section of this chapter.

4.2.4 Scheme Description: Step by Step

Having covered the major building blocks and services required for a PnP approach, this section describes the interactions and scenarios of the CORBA-based PnP scheme. However, first the scope of CORBA in this approach needs to be emphasized as follows.

The CORBA framework is used only for facilitating the installation and configuration steps of a PnP component in this design. Once the corresponding service modules are installed and activated on a client component, the client component uses the required service directly over the network without any involvement of CORBA framework. As an example, consider a PnP printer. The PnP approach uses the CORBA framework, its transport, and its services to arrange the printer device driver on the client components. Now, a client component that wishes to print a document does not use CORBA method invocation but communicates directly with the printer component over the network through the installed printer driver.

4.2.4.1 Component's layout

Consider a typical heterogeneous network. The network is CORBA-enabled; i.e., the components support the software for the ORB functionality. In this approach, a 'network component' refers to the CORBA object representation of the network device. The components are connected over a common ORB object bus, support the standard PnP interfaces, and have unique object references. The ORB is running the CORBA object services that are required for this application and these services are accessible to every component through the standard interfaces. These services include naming, trading, security, event, and persistence. Moreover, the other necessary ORB elements like Interface Repository and Implementation Repository are also available to the components.

The responsibility of maintaining the network lies on the shoulder of a network administrator. Therefore, an administration application is running on enterprise ORB that the administrator may use to configure the network components. A similar admin application, not shown in the actual layout, is also running on the vendor's ORB for managing vendor components.

4.2.4.2 PnP component's installation

Now assume that the administrator realizes the need for a new service -- say printing -- on the network. The administrator purchases a new service provider PnP component (printer) that has a built-in support for that particular kind of ORB. After its manual installation, the PnP component's software module connects the component to the ORB and finds the object references for essential ORB elements (i.e., Naming, Trading, Implementation Repository, and Interface Repository). The PnP component is assigned a persistent object reference by the ORB that is used by the component for registration with the Trader service. The component also provides the information to the Trader about the services it offers. Furthermore, the PnP component’s implementation is registered with the Implementation Repository using a POA, whereas, the component’s interfaces are stored in the Interface Repository. The PnP component also supports a configuration interface, which is used by the admin component to assign initial configuration parameters and other pertinent configuration information.

4.2.4.3 Vendor’s setup

The PnP component’s vendor also maintains an ORB for its product support. The vendor’s ORB runs the same facilities and services that are specified for the enterprise ORB. The central component on the vendor’s ORB is the vendor component, which is accessed by the vendor-manufactured PnP components. However, the PnP components require the knowledge of its vendor component for accessing it. Therefore, the vendor hard-codes the IOR (Interoperable Object Reference) and stub of its server component in each of its manufactured devices so that they are able to contact the vendor after being installed. Due to the use of persistent IOR for components, a component’s reference remains valid even after its location change. The vendor component also implements the standard interfaces for its clients (PnP components). Instead of using the static skeleton approach, the use of DSI at the vendor component is suggested because it allows a vendor to change the implementation of components dynamically.

4.2.4.4 PnP component’s registration with its vendor

After executing an initialization routine and getting appropriate references of the required CORBA services, the newly added PnP component invokes the registration interface on the vendor component using the pre-configured IOR. This registration interface may be proprietary because both the participants -- the PnP component and the vendor component -- belong to the same vendor. The registration request by the PnP component is invoked using the hard-coded static stub approach[3]. The static request goes through the component's ORB and is forwarded to the vendor ORB, where the vendor component object implementation is contacted through the Implementation Repository and the registration is completed. After registration, the PnP component may invoke a standard interface method to download the latest server-side module.

4.2.4.5 PnP component service advertisement and client's requirements

After the successful registration, the PnP component is ready to offer its service to potential clients. There are two modes for service provisioning as described in section 3.5.2.6. In the first mode (Pull), a service provider PnP component allows interested clients to discover its service, whereas the PnP component initiates the service offer and discovers its clients in the second mode (Push). These two modes can be incorporated in CORBA-based approach as follows.

Assuming that the PnP component is in charge of the service provisioning, then the question is that how does a new PnP component know which clients are interested in its service? It is proposed that each interested client registers with the trading service and

---

[3] The registration interface could as well be invoked dynamically.

specifies its requirement and a PnP component queries the Trading service and makes a list of all the objects (its potential clients) that have shown interest in its service.

The second mode is more complex. In this case, the service provider component advertises its service with the Trading service, whereas a client component queries this trading service for finding a service. If the service is not found, clients need to wait until such a service registers with the trading service. The CORBA's Event service is used to provide this notification mechanism; i.e., a 'service availability' event is defined for each type of service, whose consumers are the client components and supplier is a PnP component that offers the associated service. When the PnP component generates the 'service availability' event after completing its initial configuration routine, the registered client components are notified through the standard interfaces. The event contains the IOR of the PnP component as the event information that is used by the client components for interacting with the PnP component. Since the clients now have discovered their required service provider component, therefore, the clients invoke a standard interface on the PnP component that registers the caller client component. The PnP component waits for a reasonable amount of time (depending on network latency, size of network) after generating the service availability event before assuming that all the interested clients have been registered.

Whether using first or second mode for the service discovery and requirement, the PnP component has knowledge of its potential clients at this stage and the next step is to find client's platform (i.e., machine, OS, etc.) so that an appropriate service module may be arranged for the client component. The PnP component invokes a method on each registered client to retrieve the client's information. The method returns a structure that contains the information about the client component. Note that clients may also supply this information during their registration routine with the PnP component in first case, thereby saving an extra method call by the PnP component on each client. The PnP component now has the pertinent information about its client components, from which the component determines the requirements of its client and constructs a requirement list.

4.2.4.6 Service module distribution and activation on clients

At this stage, the clients are ready to receive a service module. Of the two possible modes of module distribution to the client components (section 3.5.2.6), this approach prefers the mode in which a vendor is responsible. Therefore, the PnP component invokes an operation on a standard module interface on the vendor component and passes the list of client components (their IORs) along with their required module types. The PnP component may use CORBA's deferred synchronous invocation mode for this operation and hence does not block for the response. The vendor component receives the request from the PnP component, gets the required service modules from its repository, and sends these service modules to the corresponding client component by invoking standard PnP interface calls and using IORs supplied by the PnP component. It is to be noted that what is required here is sending an actual executable service module to the client component and not its reference. One of the shortcomings of the CORBA framework is that it only supports the option of passing object "by-reference" and not "by-value" in method invocations[4]. In addition to the code that is needed to use the service, the service module contains the information about its corresponding PnP service component. In cases such as printer provisioning, where the service requires just a client-side counterpart, the service module may just contain executable code, whereas for some other cases, it may contain extra information about the server component so as to serve as a proxy at the client. Ultimately, the client receives this service module (as a byte stream) and activates it on its machine. In this manner, all the interested clients are configured and the vendor component sends a response back to the PnP component. With this, the initial cycle of installation and configuration of the PnP component is completed.

4.2.4.7 Dynamic configuration

For keeping up with the changing network environment, the PnP components should also configure any client components that are added after this initial configuration cycle.

If the PnP component is responsible for the service provisioning (Push mode), then it should poll the Trader service time to time, check all the client components that are interested in its service and filter out the interested clients that have not yet been configured. Once all new clients are discovered, the module distribution and activation process is repeated for these non-configured components. An alternative solution would be to use the Event and Notification service to notify the PnP component whenever some new interested client registers with the Trader service.

---

[4] CORBA 3.0 incorporates this object by-value (OBV) passing feature.

Future configuration support is implemented differently when clients are responsible for discovering their server components (Pull mode). In this case, a newly added client component registers with the Trading service and queries it for any PnP component that fulfils its requirements. If such a PnP component is found, the IOR of the PnP component is retrieved from the Trader service and then interface methods are invoked on the PnP component that would in turn call provisioning routines for the new client component. If no PnP component is found by querying the Trader service, the client component uses an event notification mechanism for registering with the PnP event channel to receive notifications when such a PnP component becomes available.

4.2.4.8 Service module upgrade

The other major requirement for supporting the PnP approach is related to the upgrade of service modules. Since upgrades may arrive anytime, an event notification mechanism is needed. A vendor already has the list of registered PnP components, therefore it notifies them directly whenever a service module upgrade occurs. The notified PnP components in turn call back the service module interface on the vendor to retrieve the upgraded service module. If the module is a server-side module, the module is installed and upgraded at the server PnP component (using the DSI mechanism). Whereas, if the upgrade is related to the client-side, then all the client components that were configured previously for that service module need to be upgraded. The PnP component may query all the affected client components for their interest in upgrade by invoking a standard PnP interface. For all those clients that have shown interest in upgrade, the PnP repeats the steps of module distribution and installation. Note that a vendor may also directly notify client components in case of client-side service module upgrade but it requires the vendor to maintain a list of all the registered clients.

Although this PnP approach has not mentioned security explicitly, it is assumed that all those interactions in this PnP application that require security controls (section 3.5.4.2) use the CORBA's Security service.

4.2.5 Conformance to the Requirements

This section looks back at the CORBA-based design of the PnP approach and tries to evaluate if the design conforms to the requirements presented in chapter 3.

CORBA allows various components to automate their installation process, where the ORB serves as the glue among PnP participants by providing many important facilities. The components, serving as the CORBA object, immediately discover the network through the help of the ORB. The trading service acts as the broker component and allows PnP components to advertise their services and clients to find their required services. The tasks of service provisioning are facilitated by CORBA's static and dynamic invocation mechanisms. The CORBA-based approach also conforms to the requirement of service upgrades and dynamic configurations by using notification facility of CORBA. The scheme also benefits from CORBA's services for Security, Persistence and Trading, thus fulfilling requirements placed in section 3.5.4.

4.3 EVALUATION

The following advantages and disadvantages have been observed for the CORBA based PnP application.

4.3.1 Advantages

1. CORBA environment suggests an architecture that runs on every platform. Therefore, the CORBA based approach provides a portable, object oriented and inter-operable application that is hardware, operating system, network and programming language independent.

2. This approach allows development of dynamic applications that can be easily assembled and modified on the fly by using DII, DSI and Trader facilities. The Trader facilitates dynamic discovery and late binding to the services as new objects can easily be added in the system and operations can be invoked easily using DII. On the other hand, the provisioning of DSI allows modifications (construction) of a dynamic server object at run time.

3. The event and notification service makes the configuration task easier for dynamic environments.

4. The use of DII and DSI also provides benefits of a smaller memory foot print since they only require a ***single*** interface for all the invocations when compared to the static mode, which requires a separate stub and skeleton per interface.

5. The CORBA framework saves a programmer from worrying about the basic services for distributed objects (directory, security, transactions, persistence, concurrency, event) because CORBA provides many object services that are common and useful for most distributed objects and applications.

6. OMA standardizes the component's interfaces that enable development of a PnP environment.

7. The clients see only the server object's interface and not its implementation, allowing the substitutability (easy upgrade) of the server implementation.

8. The client and server components are highly de-coupled, involving three intermediate elements -- IDL stub on the client side, one or more ORBs, an IDL skeleton on the server side. This isolation provides a great amount of flexibility.

9. The CORBA framework may also contribute significantly to other Network Management applications [22], [23]. Furthermore, any legacy network management application can easily be ported to the CORBA environment by giving it an IDL interface and wrapping it inside a thin CORBA object.

10. CORBA 3.0 specifications include feature of object mobility (Object-By-Value) within or between ORBs. It will provide the kind of flexibility that is often found in mobile agent frameworks.

4.3.2 Disadvantages

1. Since CORBA is a complex and expensive framework, therefore, the requirement of ORB support on each component may not be acceptable for smaller devices.

2. CORBA is not the best solution for real-time, embedded systems due to its large footprint and performance issues. Therefore, if an ORB is to be burnt into a chip, it has to shed much of its functionality. This raises the issue of trade-off between simplicity and power. It is to be noted that CORBA 3.0 specifications are making an attempt to define a Minimal/Embedded CORBA.

3. Though CORBA specifications cover many services, most of the vendor's ORBs

support only the major and essential services.

4. Dynamic invocations (DII and DSI) are complex mechanisms and not all ORBs support them. Furthermore, the DII and DSI require more effort and coding on the part of the programmer.
5. This approach still requires some expertise from administrator for maintaining the ORB.
6. Since the PnP interfaces should be standardized and supported by all components, a separate standards body is needed for the PnP approach.
7. The existence of some proprietary protocols among certain CORBA elements (mostly at the server object side) makes portability of code from one ORB to other doubtful.

Chapter 5

PnP Network Components using Mobile Agents

Traditionally, computer communication networks are based on client/server (C/S) paradigms. These paradigms typically involve stationary agents that communicate via an RPC mechanism. An alternative computing paradigm, known as Mobile Agent technology, has evolved recently to exploit the use of code mobility. Mobile Agent technology integrates many diverse disciplines such as distributed object architectures, artificial intelligence and distributed processing. Also, mobile agents constitute a computing paradigm that provides significant advantages over traditional C/S computing for certain applications. Flexibility and ease of use are the most prominent features of this technology that are well suited to any application which has to deal with a dynamic network environment. These features make mobile agents a good choice for building network management applications in general and the PnP application of this thesis in particular.

This chapter focuses on the design and implementation of a PnP scheme for network components using Mobile Agent technology. The chapter begins with an overview of the code mobility, followed by a detailed section on Mobile Agent technology, where the OMG's Mobile Agent System Interoperability Facility (MASIF) specification is described to show the concepts related to mobile agent technology [29]. After outlining the mobility framework, which is used for this implementation, the mobile agent based PnP scheme is described in detail. Finally, the chapter concludes with an evaluation of the scheme.

5.1 CODE MOBILITY

The idea of moving software programs by means of a network is not new. Since the early days of networking, hosts have been downloading applications from file servers (where they are stored) to their machine (where they are executed). Similarly, a Java applet of today's Internet is an example of mobile code, which is downloaded from a remote server and executed locally. In the above examples, code is downloaded and moved from a *storage* environment to an *execution* environment. However, this downloaded code does not usually carry the *state information* and thus starts execution from an initial entry point.

More recently, the trend has been to *"move programs while they run"* [10]. These software programs are called the 'Mobile Code' and this approach is central to a powerful computing paradigm called the 'Code Mobility'. A well-respected paper [9] describes code mobility as "*… the capability to dynamically change the bindings between code fragments and the locations where they are executed*". A conceptual framework for code mobility, as described in the above paper, is characterized by three entities -- design paradigms, technologies, and applications.

The mobile code paradigms are defined according to the location of code, data and resources, before and after the task execution [3]. There are three such paradigms -- Remote Evaluation (REV), Code On Demand (COD), and Mobile Agent (MA). In REV, the service code is sent from a source computer to a remote destination computer where it is executed using the processor and resources of the remote node. On the other hand, the COD paradigm defines a mechanism whereby the code, which is provided by the remote computer, is downloaded and executed *entirely* using the local resources and processor. Finally, in the MA paradigm, the source computer has the code and the processing capability but the computations take place on the remote computer where the resources reside. Moreover, the downloaded code may migrate from node to node, either autonomously or using the programmed destinations. To summarize the main difference between these paradigms, only code is moved in REV and COD whereas entire computational unit (code + state) is moved in MA paradigm.

To attain the code mobility feature, a programming language and a code execution environment is required. Currently, many technologies are available and being used -- Java from Sun Microsystems [40], Aglets from IBM [15], Odyssey from General Magic [11], Grasshopper from IKV++ [16], MOLE from the University of Stuttgart [51], AgentTcl from Dartmouth College, Obliq from DEC, Concordia from Mitsubishi, and several others.

Of these design paradigms and technologies, this implementation chooses ‘Mobile Agents’ as the design paradigm, Java as the implementation language, and a proprietary code execution environment (section 5.2.4).

5.2 MOBILE AGENT TECHNOLOGY

The word agent is used in many areas and thus bears many definitions. In the context of this thesis, the word is qualified more by calling it a *software agent*. There are many ways that we can define software agents and their types, however, this thesis has adopted the definitions proposed in the OMG's MASIF specification [29].

Therefore, a software agent is *a computer program that acts autonomously on behalf of a person or organization.* Software agents are divided into two main categories -- stationary and mobile agents. A stationary agent *executes only on the system where it begins execution* and uses an RPC mechanism to interact with other agents and the system. A mobile agent, however, *is not bound to the system where it begins execution and has the unique ability to transport itself from one system in the network to another*. In other words, a mobile agent is a running computation (procedure and data) that can move from one host to another host over the network in order to carry out its operations. Like a CORBA object, a mobile agent is also a distributed object that not only has a behavior, an identity, and a state but also a location. A mobile agent, moreover, has the migration capability and so it can be located on different machines at different times during its life cycle.

Like typical client/server architectures, mobile agent based applications also have a client and a server side, however, the difference lies in how the two communicate. When the client in the mobile agent architecture needs data or access to a resource that the server provides, the client does not talk to the server over the network; instead, the client sends a mobile agent to the server machine. Since both the state and data of the agent are transferred, the agent not only remembers what to do and how to do it; it also remembers what it was doing before.

5.2.1 Mobile Agents Applications

Mobile Agent technology has applications in various domains. Typical application domains are distributed information retrieval, telecommunications service management, remote device control and ***configuration***, electronic commerce, active networks, real-time interactions [4]. This thesis work, however, explores this technology for configuration management of components in networks.

5.2.2 MASIF

Since mobile agent technology is relatively new, there are no widely accepted standards. There are several proprietary mobile agent systems available (mentioned earlier) that differ widely in their architectures and implementations. This raises the issue of interoperability among different agent systems. The lack of standardization and interoperability are the biggest issues that are impeding the growth of mobile agent technology. Therefore, standards are needed for a universal agent platform that would allow any server to accept and execute an agent from any vendor. The OMG is working on agent standards as a CORBA common facility for defining the agent interactions based on language-independent interfaces. MASIF, originally called the Mobile Agent Facility (MAF), is one such mobile agent standard that was accepted as an OMG technology in February 1998.

The MASIF specification [29] defines a common conceptual model for mobile agent systems by introducing and defining mobile agents and their related terminology. The specification does not define the standards for local agent operations or interoperability between system languages; rather, it focuses on the interoperability between agent systems. The goals of MASIF are to:

- enable interoperability between agent platforms of different vendors;

- integrate the RPC paradigm and mobile agent technology;
- benefit from the existing CORBA standards.

The current standardization topics addressed by MASIF include:

- agent life cycle management
- agent transfer and tracking
- naming of agent and agent systems
- agent system types
- integration of common CORBA services
- agent system security

A closer look at the MASIF specification identifies three main components -- an agent system, agents, and a communication infrastructure. MASIF defines an *agent system* to be a platform that can create, interpret, execute, and terminate agents, thus serving as a host for agents. Just as the distributed objects within a CORBA-based system rely on ORBs for most of their functionality, mobile agents require an agent system (host) to function. In addition to a name and an address that uniquely identify an agent system, the agent system is also associated with an *authority* that identifies the person or organization for which the agent system acts. Moreover, each agent system has a *type* that describes the profile of the agent system, thus implying support for a specific types of agents. The clients (source systems) use this profile to check if the destination system is capable of supporting mobile agents of the source system's type.

A set of mobile agent systems that have the same authority forms a *region*. This concept of region introduces a need for a *region administrator*, who is responsible for maintaining the region network. In a big network, there may be many regions that may interconnect and inter-operate using a shared naming service. However, this PnP application is simulated in our lab environment and thus constitutes a single region.

Each agent has an *authority* and a globally unique name much like the agent system itself. An agent also has an associated location (address) that specifies the agent system where the agent resides. The MASIF specification defines the concept of a *place* -- a context within an agent system where an agent can execute, providing useful functions such as access control. A client uses the address of a region[1] along with the name of an agent/place to communicate with an agent/agent system respectively.

The communication infrastructure provides services for common transport, naming, and security for an agent system. Using the communication transport, a mobile agent may travel from one place to another, where its state and code are transported. The state of an agent can either be its entire execution state (i.e., stack, threads, attributes, etc.) or just the value of its attributes. These types of mobility are called *strong* and *weak mobility* respectively. For agent migration, the source agent system requests that the agent travel to the destination agent system and if the destination agent system agrees to this transfer, then the agent's state, authority, security credentials, and code (if necessary) are transferred to the destination agent system. After transfer is completed, the agent is reactivated and its execution is resumed.

An agent class specifies the interfaces and the agent system uses these interfaces to

---

[1] Actually the address of an agent system that serves as a region access point.

control the life cycle of the agent. The agent system, on receipt of a visiting agent, spawns a separate thread where the agent's class is instantiated and executed. Other important functions of agent systems are related to finding mobile agents/systems and enforcing security.

MASIF also specifies the use of many CORBA object services. For example, the Naming service can be used for finding mobile agents and agent systems. Though, the CORBA Security service does not meet all the needs of the mobile agent technology, MASIF implementations may use available security features to satisfy their needs. CORBA's Externalization service is used for saving/restoring a mobile agent's state to/from a data stream. If a mobile agent is represented as a CORBA object, it may also take advantage of the CORBA Life Cycle service for its creation, deletion, copying and migration operations.

5.2.3 Java and Mobile Agent technology

One of the basic requirements for a mobile agent based system is the presence of an infrastructure that supports the aforementioned mobility facilities. However, supporting these facilities leads to certain requirements for the mobility infrastructure:

1. a mobile agent should be able to execute on any machine within a network regardless of the processor type or operating system used;
2. an agent's class should not have to be installed on every machine that the agent could potentially visit; rather there ought to be a mechanism to download the agent class from a code base whenever required;

3. interactions among agents and agent systems are to be provided in a safe and secure way.

The Java programming language provides a solution to these requirements. The power of Java comes from its facilities for code migration (dynamic class loading), security, platform independence, and advanced networking capabilities. Since Java ensures safe and secure execution of mobile code independent of the underlying executing machine, therefore, it is one of the most popular programming languages used for building mobile agent systems.

Having described the role of Java, it is time to choose a suitable agent system (mobility infrastructure) for implementing the proposed PnP approach of chapter 3. The agent system used for developing this PnP application is called the ‘Mobile Code Toolkit’ [34],[49]. Before discussing the plug-and-play implementation details, an understanding of this mobile agent system is important.

5.2.4 Mobile Code Toolkit

The Mobile Code Toolkit (MCT) is a mobile agent system that has been developed under the umbrella of the Network Management research group at Carleton University. Like many other mobile agent systems, the MCT software is written 100% in Java. Figure 5.1 shows the architecture of this mobile agent system. The high level view of the system illustrates three major interacting components:

- Mobile Code Daemon (MCD)
- Mobile Agents
- Interface to network resources (VMC)

The Mobile Code Daemon is the principal component for providing mobility and agent transaction facilities and thus serves as an agent host (agent system) in MASIF terminology. The MCD is a thread that runs under a JVM and listens on a known TCP/UDP port for an agent arrival. The other agent hosts that are of the same type (i.e., also running a MCD) can communicate and send an agent as compressed byte code over a TCP/UDP connection to the receiving agent host. The concept of location (address of agent system/host) is simply implemented as a socket number for the agent host and is called a *migration target* in this agent system. The MCD provides the facilities that are defined for the agent host of an agent system in MASIF specification. These facilities include byte-code management, communication, instantiation and migration of the agent, and security. Due to the importance of security, the MCT addresses security risks by complying with strong security standards like DSA, MD5, and SHA-1.

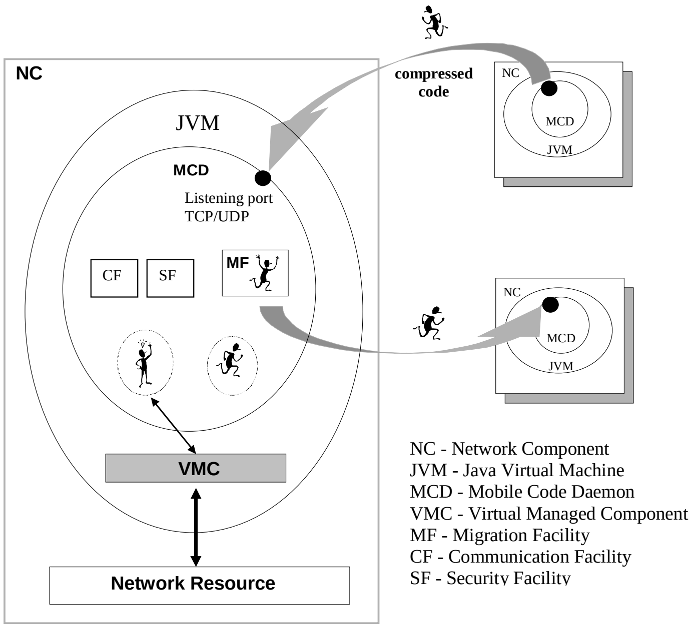


Figure 5.1: Mobile Code Toolkit

The toolkit implements a mobile agent as a Java class that can be sent over the network in the compressed code format. Each mobile agent in this system has an associated authority and a unique name as per the MASIF specification. When a mobile agent arrives at a host, it needs to access the host resources to perform the network management tasks. For this purpose, each mobile agent is provided a context (i.e., a place) to execute and is not allowed to access the host environment and resources directly. Therefore, an *interface* to each network resource is provided on the individual network components (hosts) which is called the Virtual Managed Component (VMC). An agent can use this interface to access the required resources on a component.

The profile of our agent system can be specified in MASIF terminology as follows. The name of the mobile agent system is the Mobile Code Toolkit; its *type* is MCT; it is implemented at Network Management Lab of Carleton University; it supports Java as the implementation language, it uses migration targets for travelling, etc.

5.3 MOBILE AGENT BASED PLUG-AND-PLAY APPROACH

Having discussed the mobility infrastructure in the previous section, the stage is set for describing the mobile agent based implementation of the proposed plug-and-play approach. The description starts with a high-level view of the approach, followed by a detailed explanation and analysis of the implemented approach.

5.3.1 High-level Scenario

Consider a network of several Java-enabled components that support MCT (the agent system) and an appropriate plug-and-play negotiation protocol. The vendor supplies a PnP component that is also Java-enabled with a pre-installed plug-and-play interface.

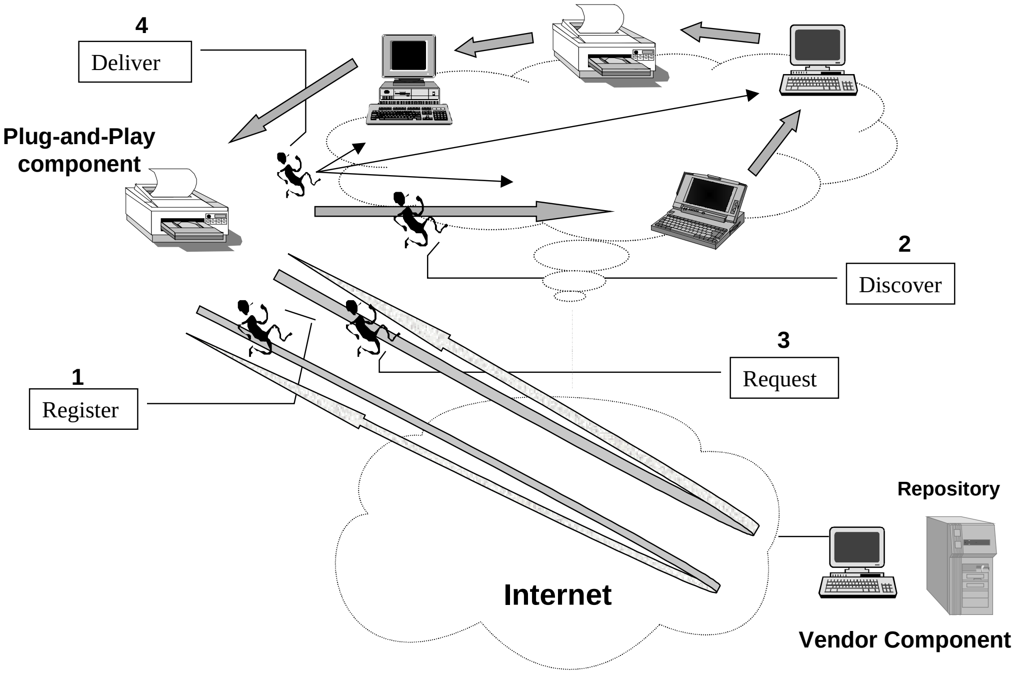


Figure 5.2: Mobile Agent based Plug-and-Play scenario

This interface not only provides access to the component; it also initiates and controls the auto-installation process by sending and receiving the mobile agents.

Figure 5.2 illustrates the simple scenario of the mobile agent based plug-and-play scheme. First, the PnP component is plugged-in and attached to the network. As soon as it is turned on, its bootstrap program installs and activates the software module, which is capable of sending mobile agents within the network. To begin the configuration process, a mobile agent is dispatched to the vendor site for the ***registration*** purposes (#1 on Fig. 5.2). After successful registration, another mobile agent is sent over the network to ***discover*** the configuration requirements of other network components (#2 on Fig. 5.2). This mobile agent gathers information (such as, component’s type, OS, other attributes) from each component and then proceeds forward to the next node in the network using the default migration targets. When the mobile agent returns to its originator, the PnP component's software determines the configuration requirements of the supported network and sends another mobile agent with these requirements (i.e., ***request*** for modules) to the vendor site (#3 on Fig. 5.2). This request agent arrives at the vendor component and accesses an interface to the service module repository. If we make the PnP component responsible for the distribution of service modules (section 3.5.2.6), then the requested modules are sent back to the PnP component. After that, separate dedicated ***delivery*** agents are dispatched to each discovered component, carrying the appropriate service modules (#4 on Fig. 5.2). These mobile agents contact the set-up interfaces at each component and deliver the corresponding modules to them. These modules may be installed immediately or stored locally at the destination component. This completes the cycle of initial installation and configuration of the PnP component and other network components. Now, the PnP component is up and running and other components are ready to use it whenever required. For future support of the PnP component, two more steps need to be completed. First, a permanent mobile agent must be injected into the network that keeps scanning the network for future discoveries (e.g., addition of a new client component that needs to be configured for the service). Furthermore, the vendor notifies all the registered PnP components through a mobile agent on any service module upgrade in future. The PnP component, in turn, makes the necessary arrangements for its upgrading or the corresponding client components upgrades.

5.3.2 Mobile Agents or Messaging: Which way to go?

The simple scenario presented in the previous section used mobile agents as a means of communication among components. However, in some cases, a simple messaging mechanism could have been used to achieve the same task. Following are some conclusions for selection between the alternatives of mobile agents and messaging.

Mobile Agents are *smart* messages that have behavior associated with them, whereas a message just has passive data with no active behavior. This scheme would use mobile agents in all situations where the messages are required to have some behavior. Furthermore, mobile agents are a good choice when asynchronous communication (e.g., when a delayed response is expected/possible) is needed, or there is one sender and many recipients for the message. The other important use of mobile agents is in the area of remote information searching and filtering, where they may help in saving network bandwidth. On the other hand, the drawbacks of mobile agents include security issues, consumption of significant computational resources due to the Agent Execution Environment (AEE) and the overhead of an agent class in addition to the actual message.

The Messaging uses a socket as a communication channel between a client and a server. Although both the mobile agents and the messaging have to carry extra data[2] in addition to the actual message, the messaging incurs lesser overhead when compared to mobile agents. Messaging is appropriate for simple, plain, one-way (possibly two-way) messages. The major drawbacks of the messaging mechanism are that it has poor security too. Furthermore, an application may not always be allowed to create a socket connection

---

[2] Extra data means agent’s state and the message headers in case of mobile agents and messages respectively.

due to firewalls.

In addition to the above conclusions, the use of mobile agents as the communication entities is preferred over messaging in this application for two main reasons:

1. Mobile agents are easier to program and customize and provide more flexibility in this PnP approach.
2. Since a MCD (agent host) is already running on each component and listening on a port for agent arrivals, the approach can take advantage of this existing channel of communication by sending agents instead of creating a new socket for messaging.

5.3.3 General Setup

The mobile agent based plug-and-play scenario is implemented in our Network Management Laboratory by setting up a test environment and using a simulated printer as an example of a PnP component. In this case, a PC was used to represent the printer's functionality. The simulated network comprised of the following three interacting components (section 3.4.1):

1. Plug-and-play component (printer)
2. Vendor component
3. Client components

It is to be noted that both the PnP and client components are network components, where a PnP component offers a service and the client components are other network components that may use this service. These components are represented by various PCs from different vendors (running Windows NT/95) and the Sun Workstations (running

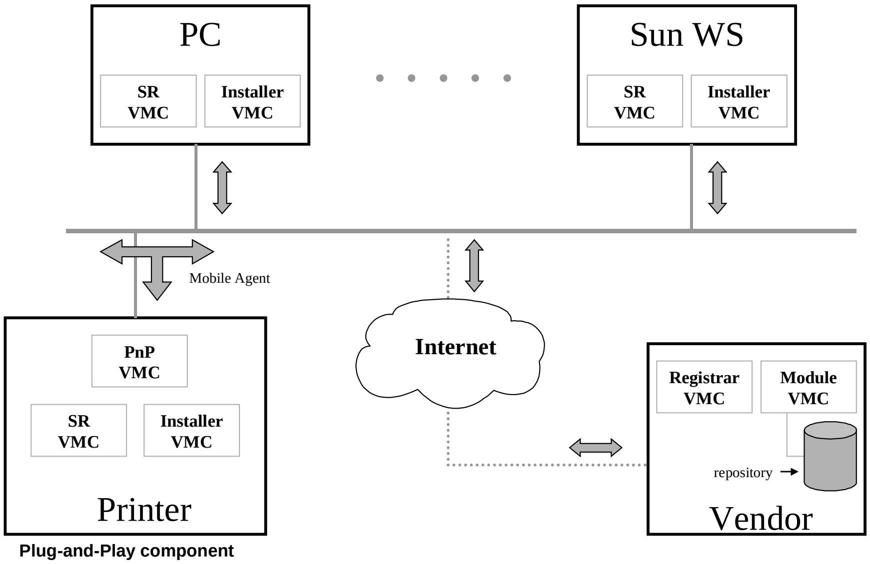


Figure 5.3: Setup of example network

Solaris) that make the simulated environment heterogeneous in nature. Figure 5.3 shows this example network of components along with the implemented agents and the VMC interfaces. The abbreviations and functions of these VMCs are explained later.

5.3.4 Assumptions

In chapter 3, a general PnP approach and its major requirements were described. For fulfilling these requirements, as well as to keep the implementation simple, the following assumptions are made in this mobile agent based scheme:

- Each network component is Java-enabled and acts as an agent host by running the agent system (MCT). This corresponds to the software requirement presented in section 3.5.2.1. Since it has been realized that the assumptions regarding running Java and an agent system on a network component is a bit unrealistic today, therefore, this scheme suggests a minimal set of mobility functions and a JVM for small components. Moreover, this approach expects that the network components will be Java-enabled in the future with the advent of Java chips. There are already some examples of Java-enabled component present, which strengthen this belief. One such example is Sun Microsystems' Java Card ™ technology that enables Java to run on smart cards and other devices with limited memory.

- The PnP component is shipped with a vendor-supplied plug-and-play module called the PnP-VMC. This component knows the address of the agent interface of its vendor through the information pre-configured during manufacturing (software requirement of section 3.5.2).
- All the components belong to the same authority and form a single region, thus establishing better degree of trust.
- Bi-directional migration paths are already established among the existing network components.
- Migration paths are configured on the components to form a *single* complete ring.
- The plug-and-play negotiation protocol, the interfaces, and the names of the VMCs are standardized.
- A directory server (serving as a lookup server for components and agents), a DHCP server, and a MCT boot server (from which a MCD can be downloaded) are available in the region. Note that, the use of a directory server in our mobile agent system conforms to the requirement for a service broker presented earlier in section 3.5.2.6.
- An administrator is responsible for specifying the requirements of the network

components, providing initial configuration parameters and controlling security issues.

5.3.5 Implementation

The mobile agent based PnP application was implemented using Mobile Code Toolkit infrastructure (section 5.2.4) and Java. The application developers, that use the Mobile Code Toolkit, are required to write and customize the mobile agents and the network component's VMC interfaces. Analogous to static method invocation in the CORBA framework, the VMCs consist of published interface methods that are invoked by the visiting agents. Therefore, two kinds of modules are developed in this PnP application:

1. **Mobile Agents**: to be sent to various network components with specific delegated tasks;
2. **VMC interfaces**: to provide access to the actual network resource and component.

The MCT provides a generic structure for an agent that can be customized for implementing various agents with different delegated tasks.

The VMCs are required on all the network components that are the potential recipients of mobile agents. A network component can have more than one VMC installed if it provides different interfaces. A mobile agent can access a VMC by using its standard name. Although, the current implementation uses function-oriented names for the VMCs, the standardization of this scheme may define other standard names in future. Five different VMCs are implemented that are distributed over four different interacting components.

The implemented PnP application comprises 12 Java classes and approximately 1700 lines of code. Appendix-A shows class hierarchy of the implemented scheme and a screen-shot of running application.

5.3.6 PnP Scheme: Step by Step

Figure 5.4 presents interaction among various components of this mobile agent based scheme. The diagram exhibits the transactions among various agents and VMC interfaces by illustrating the dispatched agents, inter-acting participants, and the receiver VMCs.

The following sub-sections discuss the installation and configuration steps of this mobile agent based plug-and-play scheme (section 3.5.1) in detail. These sections describe what is actually implemented and what are some alternative implementation strategies.

5.3.6.1 initial installation, booting and configuration

The vendor-supplied PnP component is Java-enabled with a plug-and-play VMC (*PnP-VMC*) pre-installed on it. This also demonstrates the fact that this approach is capable of supporting vendor-specific VMCs as well. The PnP-VMC is more powerful in comparison to the simple VMC introduced earlier and it plays an active role in the scenario by initiating and controlling the entire auto-installation process. The VMC is responsible for injecting, controlling, and interacting with most of the mobile agents. It also synchronizes agent creation, dispatch, arrival, and destruction. These responsibilities are carried out in conjunction with the MCD that runs as the underlying mobility daemon (agent host). To gain better control in the multithreaded environment of the MCT agent system, the PnP-VMC is also implemented as an independent thread. Many data structures that are used throughout the configuration process are maintained by this PnP-VMC.

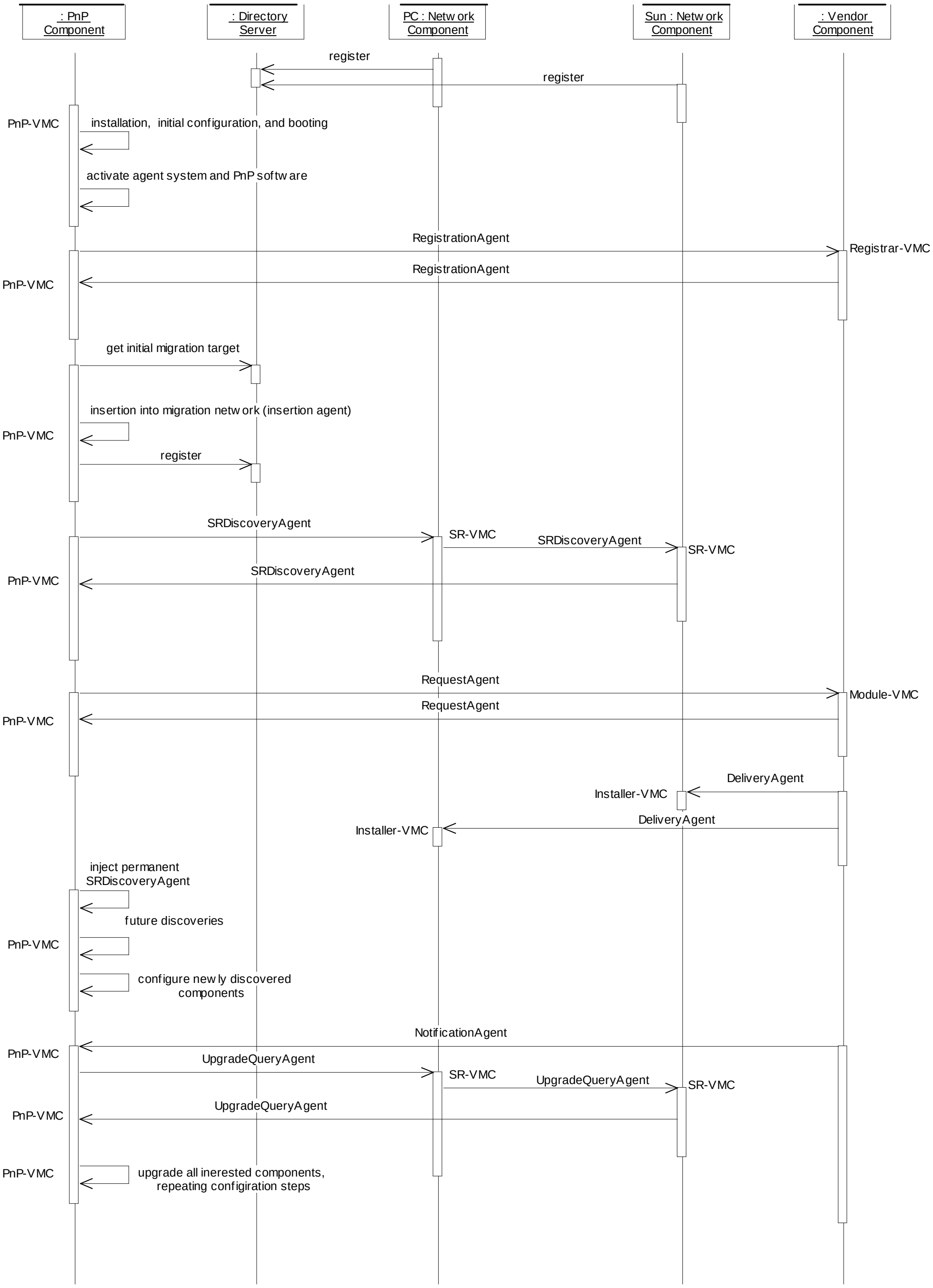


Figure 5.4: Mobile agent based PnP interactions

The first step in the scheme is to establish a component hardware connection. Having connected and before initiating the entire installation process, the PnP component needs a bootstrap program for activating its agent system. A related and more critical issue is an initial IP address requirement for the component so that it may communicate over the network. It gives rise to the question of how a component acquires its IP address and bootstrap code. The use of the Dynamic Host Control Protocol (DHCP) [18] is suggested as one of the possible solutions to this problem. Therefore, a PnP component broadcasts its request for initial configuration over the network and a DHCP server recognizes the request and assigns an IP address on a lease or a permanent basis, depending on the type of request. The server may also provide other interesting options like host name, DNS server address, sub-net mask, and boot file address. The component can now download the bootstrap code and the mobile code daemon (i.e., MCD), if it is not pre-installed, using the TFTP mechanism [17]. Note that the DHCP has provisions for supplying vendor-specific options that can be used by an administrator to supply initial configuration parameters. The major problem associated with the DHCP configuration mechanism is that it does not update the DNS. However, a recent IETF draft [19] has resolved this problem too. Thus, the issues of bootstrap code and initial configuration are addressed using the DHCP.

Since hardware network components are simulated by a software implementation in this scheme, therefore most of the aforementioned issues do not arise. However, the issues that arise and need to be addressed are resolved through the use of property files and software parameters.

After getting the initial configuration parameters and required bootstrap code, the component's bootstrap program installs and activates the MCD. This allows the vendor-supplied PnP-VMC to be activated and gives the component the capability of sending and receiving mobile agents. Now the PnP component is up and running (i.e., ready to start the configuration phase).

5.3.6.2 registration with the vendor

Using the pre-configured vendor address, the component's PnP-VMC sends a mobile *registration agent* to the vendor component for the purpose of registration. At the vendor component, an interface called the *Registrar-VMC* is provided to allow a PnP component to register using the incoming registration agent. The Registrar-VMC maintains the information about the registered PnP components in its database. The other task of this VMC is to provide upgrade notifications to all of the registered PnP components. This feature is not supported in the current implementation but it can easily be implemented by enhancing the capabilities of this VMC. After registering at the vendor site, the registration agent returns to the PnP component with the registration status. If the registration is not successful, the plug-and-play configuration process is halted and the intervention of the network administrator is required.

5.3.6.3 insertion into the migration network

This mobile agent based scheme depends largely on the migration network that has to be established among network components. Our agent system specifies agent migration APIs that allow an agent to migrate either to a *specified* target or to a *default* target. The default migration targets of the components can be manipulated in such a way that the existing components form a bi-directional, ring-shaped migration network. An agent injected into this migration network can easily traverse one, many, or all of the host components. However, if a network component wants to be discovered (using mobile agent technology) by other existing network components, it needs to be a part of the existing migration network. This means that there is an additional requirement of inserting a PnP component (in fact, all the network components) into the established migration paths of the existing network components.

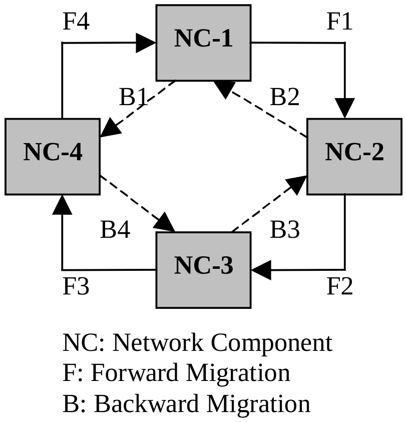


Figure 5.5: Migration targets

This problem can be solved easily if we can obtain an initial migration target[3]. The PnP component dispatches an agent, called the *insertion agent*, to this initial (specified) target. After migrating successfully to an initial target, the agent can simply use the configured migration paths (default targets) for further migrations. The forward migration of this agent using the *existing migration targets* is based on an earlier assumption regarding the establishment of migration paths among components, as shown in figure 5.5. The task delegated to an insertion agent is to insert newly added PnP components into the existing bi-directional migration network in a linked-list manner. The agent implements a simple and secure mechanism to perform this task. It moves from its source PnP component to an existing host on the migration

---

[3] i.e., the address of an agent host on the migration network.

network (initial target). After that, the agent uses default migration targets to move from initial host to next host and so on, until it reaches the last host in the ring. During its migration, the agent modifies the backward and forward migration rings (one at a time) on each visited host. After completing the delegated task, this agent commits suicide and terminates. Note that an agent is used here instead of a message to achieve the required task. An agent was chosen since this task involves one-to-many interactions and requires the navigability property.

The problem of insertion of a new component is resolved using initial targets but it raises another issue; i.e., how does a component receive this initial target. Provisioning of a directory server is suggested in the PnP system as one of the possible solutions to this issue (section 3.5.2.6). This directory server would act as a broker for the components, much like a CORBA trading service, where a component not only can announce its presence but can also find other components. A new PnP component may locate the existing migration network by obtaining an initial target (the address of a network component) through a query to this directory server. However, all the components must first know how to locate the directory server itself. The solution is to provide the address of the directory server and other basic services such as a boot server, at installation time, either using DHCP or through the administration application. Since our agent system (MCT) does not have a true directory service, this PnP implementation has resolved this problem by passing the boot time parameters through property files.

5.3.6.4 discovery of clients and configuration requirements

By now, the PnP component is ready to offer its service to any client component. As stated in section 3.5.2.6, there are two possible modes for service advertisement and discovery. In Pull mode, the clients are responsible for finding the service, whereas, in Push mode, it is the PnP component that is responsible for finding its potential clients and pushing (offering) its service.

For implementing the Pull mode, it is suggested that all the PnP components register their services with a centralized directory service and all the interested clients look up their required service in this directory. This implies the use of an event notification mechanism to handle the cases when the clients do not find the required service in the directory. A client must register with an event service that would be responsible for sending a notification to the client when a PnP component, which offers the required service, registers with the directory. Since our mobile agent system (MCT) does not support a distributed event facility currently, this mode is not implemented.

The Push mode implementation for service discovery requires the use of dedicated mobile agents. The PnP-VMC injects a mobile agent, called the *service requirement discovery agent (SRD-Agent)*, into the migration network for discovering the configuration requirements of the other network components. Based on the assumption made earlier on, it is the responsibility of an administrator to set the requirements of each individual component. Therefore, the administrator sets an attribute for each newly installed network component (client) that specifies the service type required by the component. On the other hand, the SRD-Agent has the information that defines the service type offered by its source PnP component. The agent migrates from one component to another, contacts a service requirement interface (*SR-VMC*), and checks the visited component’s required service type against the offered service by using the service matching mechanism described in section 3.5.2.6. If the visited component's requirements match the service that the agent’s source PnP component is offering, the agent gathers information regarding the component type and OS from the visited component. This information assists in determining the configuration needs of the component. Access to this information is also provided through the SR-VMC that maintains it in an information database (much like a MIB). After traversing the entire network and having gathered sufficient information from all the interested components through their SR-VMCs, the SRD-Agent returns to its source PnP component. This marks the end of the requirement discovery cycle.

There are some issues associated with the scenario described above. First, there are many possible ways to relay the gathered information back to the PnP components:

1. The agent accumulates the information that is gathered from the visited components and communicates this information to the PnP component only when the agent arrives back at the component. The problem with this approach is that the agent’s size may grow with each hop, thus introducing overhead in agent transfer and management. Furthermore, the integrity of the information is more at stake when the agent traverses the entire network while carrying this information.

2. An alternative method is to send the information collected from the currently visited component immediately over a socket connection to the PnP component. While this approach is simple and solves the problems of the first approach, it requires agents to be authoritative enough for creating a socket at each visited host. The other problem is

that socket-based communications are relatively less secure when compared to mobile agents. This issue can be addressed using Secure Socket Layer (SSL) protocol that has been proposed and adopted by Netscape Corporation in its web browsing applications and already forwarded for Internet draft.

3. Another approach, constituting a middle solution, can also be proposed. Here, an agent keeps accumulating the information until some pre-defined threshold is reached, after which the information is sent back to the PnP component through an agent. To send this intermediate information, the SRD-Agent clones an *information agent* that is sent back to the PnP component while the SRD Agent continues forward in the network. Since the SRD Agent has to clone other agents, it must carry byte-code for that agent class, causing significant overhead. Of the three suggested approaches, the second approach (i.e., socket based) is chosen in this implementation.

Another important issue at this configuration phase is regarding the use of the initial SRD agent; i.e., what to do with this agent after it completes its initial discovery of the client components. This issue is highly related to the mechanism used for future discovery of client components and is discussed in a later subsection.

This subsection has presented the mechanisms for discovering the service requirements of the components in the network. These features of navigation, communication between a PnP component and many network components, and remote searching (possibly, with filtering) present a scenario where mobile agent technology is best suited. This is why the mobile agents are used for this discovery cycle.

5.3.6.5 service module distribution and activation

Having discovered the client components, the next step is to arrange service modules for the clients. The approach where the vendor is responsible for the distribution of service modules (section 3.5.2.6) is discussed first.

From the accumulated information about the client components, the PnP component determines the configuration requirements and composes a list of addresses of client components along with their required service modules. The list is contained within a new mobile agent, called the *request agent*, which is dispatched to the vendor component. Although it is admitted that the messaging would have been a better alternative here (due to the requirement of passing just a passive data), still a mobile agent is used in the implementation for the reasons mentioned earlier in section 5.3.2.

The vendor provides a *Module-VMC* as an interface to its repository of service modules that a request agent can access for requesting the required modules. The vendor's Module-VMC takes the list from the incoming agent and collects the corresponding service modules from its repository. The vendor component constructs the dedicated mobile agents for each discovered client component and dispatches them to their destinations. These mobile agents are called the *delivery agents* and the task delegated to them is to deliver and distribute a service module to a client component. Each network component maintains an interface, called the *Installer-VMC*, which accepts an incoming delivery agent and detaches the corresponding service module from the agent for set up and activation purposes. The choice of the installation and activation mode of these service modules is left up to the programmer of the VMC. This implementation, however, supports the mode in which the VMC *saves* the incoming service module in the local file system and prints a module arrival message on the administrator console.

An alternative approach for the module distribution is also discussed in section 3.5.2.6, where PnP components are responsible for the module distribution. In this case, a PnP component only sends the list of required service modules without the addresses of client components to the vendor. Using the list, the corresponding modules are delivered by the vendor to the clients via PnP component using a socket-based communication or a mobile agent. The use of socket-based communication is suggested here since the message is point-to-point and constitutes plain data without any behavior. Having received the service modules, the PnP component distributes these modules using the delivery agent, similar to the first approach. Although this approach makes the PnP component the central component in the configuration scenario, it is inefficient because it incurs the extra overhead of downloading the service modules (from the vendor to the PnP component) and requires sufficient memory at the PnP component side.

5.3.6.6 auto discovery of newly added components in future

At this point, the initial installation, configuration and reconfiguration phases have been completed for the *existing* network component. However, due to the dynamic nature of networks, newly added components (requiring the services of the PnP component) must be discovered and configured immediately. With reference to the future configuration requirement (section 3.5.2.7), there is a need for a permanent discovery agent on the network that will keep roaming from one component to another, sniffing for any new component and notifying the PnP component in case it finds some new component. This permanent SRD agent can also be useful if an existing *configured* network component wants to negotiate with the PnP component. The agent is a prototype of the service discovery agent described earlier, however, the nature of its task is permanent and thus it never terminates.

Since the roaming agents may consume significant network bandwidth, their frequency and number must be regulated. For frequency control, it is assumed that the permanent SRD agent does not roam all the time; rather, it navigates the migration network periodically (say, six times a day) and sleeps on a designated agent host (most appropriately, the originating PnP component) for the rest of the time. The number of permanent discovery agents depends on the way they are incorporated in the scheme:

1. **Component SRD agent:** Assuming a dedicated SRD agent for each PnP component yields too many permanent live agents in the region.

2. **Regional SRD agent:** The other extreme is to assume that there is a single permanent SRD agent that fulfils the needs of all the PnP components in that region (i.e., the migration network). In this case, each PnP component needs to register with this *regional SRD agent* so that whenever a new potential client component is found, the corresponding PnP component is notified. Since such a regional component represents many service provider PnP components, it must maintain a list of the offered services along with the addresses of the PnP components that are to be notified. This list may become very large in a big network, incurring large overhead on the agent side.

3. **Service SRD agent:** It is suggested to use another approach that lies between above two extremes. In this case, a single SRD agent is defined for each type of service (*service SRD agent*). For example, the print service has only one representative SRD agent in the region with which all the PnP printers register. Similarly, a routing service has its own representative SRD agent, and so on. It is the responsibility of the first PnP component of a particular service type to inject this representative service SRD agent into the network. The service SRD agent advertises its type (service representation) by registering with the centralized directory service and all the concerned PnP components that offer the corresponding service register with this agent on its next visit.

Now we can come back to the discussion of the initial SRD agent that was deferred in section 5.3.6.4. Having introduced a permanent discovery agent, it is reasonable to suggest that each PnP component dispatch a dedicated SRD agent that would do the service requirement discovery for the PnP component for the very first time. This SRD, however, is destroyed after doing this initial bulk discovery -- either terminated by the PnP component or it terminates when it meets the corresponding *service SRD permanent agent*. The permanent service SRD agent then carries out any future discoveries.

The permanent SRD agent keeps traversing the network periodically. When it finds an interested component that has not yet been configured (possibly a new component), it sends the relevant information to the corresponding PnP component that may repeat the same configuration scenario. This way, future discoveries are controlled in a dynamic manner through the use of permanent discovery agents.

5.3.6.7 upgrade notifications from the vendor

In case of a new release of a service module, the vendor may send dedicated mobile agents to all the registered PnP components. These agents are called the *notification agents* and they contain information regarding the service module that is upgraded. For the reasons discussed earlier, mobile agents are chosen over messaging even though the messaging is a better option here. Furthermore, note that the upgrade offer may be free of charge or may involve some trading service (section 3.5.4.4). When a PnP-VMC receives a notification agent from the vendor, it first checks the type of upgrade. In case of a server-side upgrade, the PnP-VMC upgrades the service module on the PnP component using a hot-swapping technique mentioned in section 3.5.2.6. However, if the upgrade belongs to the client side, the PnP component needs to ask previously configured client components for their interest in the upgrade. This task is best achieved using a single mobile agent, called the *upgrade query agent*. This agent is injected with a pre-defined traversal list that is made up of previously configured components only. The agent migrates from a component to the next component, while querying the visited components about their interest in the upgrade. The agent completes its query cycle and eventually reaches back to the PnP component with a list of all the clients that replied "yes" for the receipt of the upgraded module. Note that, this agent behaves like a discovery agent as it traverses many components and queries for information. However, unlike a discovery agent, it does carry the accumulated information during its migrations. The reasons for choosing this approach are as follows. First, the overhead of creating socket connections for sending simple query result cannot be justified. Moreover, since the agent visits fewer components and carries little information (only the upgrade list), thus the size of the agent is not significantly increased. Also, the information carried is not confidential and we may ignore the security issues.

After getting the list of all the interested clients, the PnP component makes the arrangements for the upgraded service modules. For this to happen, the PnP component repeats the previously described configuration scenario by sending a request agent to its vendor so that the corresponding upgraded modules can be distributed accordingly.

5.3.7 Mobile Agents and Security

Before concluding the description of mobile agent based PnP implementation, the biggest concern associated with mobile agent technology -- security -- needs to be explained. This is the reason that most of the existing mobile agent systems and frameworks pay special attention to the security risks and threats. Security in mobile agent based systems constitutes a major area of research and several good papers (e.g. [8]) have been written to address these issues.

There are two aspects of security in a mobile agent system:

1. ***Agent host security***: since the hosts provide access to the local machine and resources, a malicious mobile agent can cause chaos if it is not regulated and controlled through the security mechanisms.
2. ***Agent security***: this relates to protecting an agent from a malicious host and other potential attackers.

In order to overcome the above security areas, trust must be established between an agent and host. The host must be able to recognize an agent's identity and should enforce the constraints (typically, ACLs) on what the agent is allowed to do. The resources that an agent host must protect are -- OS, file system, disk, CPU, memory, other agents, and local programs. Similarly, an agent must also insure that the host it is visiting is not malicious. While the issue of host security can be solved using standard security techniques (like Java’s security APIs), agent security is a harder problem to solve because an agent needs to give full control to its host during execution.

In our mobile agent framework (i.e., the Mobile Code Toolkit), the problem of host security is addressed through digitally signed agents and VMCs. Agent security is, however, not supported in this framework.

5.3.8 Conformance to the Requirements

Having completed the description of the mobile agent based scheme, we are in a position to determine whether the requirements presented in chapter 3 have been met successfully by this mobile agent based PnP approach.

The most pertinent requirements mentioned in section 3.4.2 have been achieved successfully in this mobile agent based scheme. For example, the components are able to auto-install after a hardware connection has been established and initialization has taken place. The use of a directory server allows components to discover and join a migration network, where the directory server acts as a broker between the components and the network. Moreover, the service provisioning and requirement determination phases are automated using mobile agents. This scheme also accomplishes the important requirement of supporting future service upgrades and dynamic configuration and conforms to the detailed requirements covered in section 3.5.2.

Finally, section 3.5.4 had proposed many facilities for supporting PnP application. Of these facilities, this mobile agent based scheme does not support trading, persistence, and notification capabilities due to the limitation of underlying MCT agent system.

5.4 EVALUATION

5.4.1 Advantages

1. It is a platform-independent scheme since it uses Java both for the mobility framework and the developed applications (agents and VMCs).
2. The added component (PnP component) initiates the installation process and eliminates the requirement of polling by the operating system, which is the case with the plug-and-play peripherals of PCs.
3. The auto-discovery of new client components and their configuration is facilitated due to the presence of discovery agents on the managed network.
4. Updates and modifications to the service modules are brought from the vendor and installed in the network automatically.
5. The use of agents allows for the injection of new functionality at runtime. For example, our scheme enables the extensibility of component behavior through special mobile agents, called extlets [49].
6. Updating agents is facilitated since obsolete agents can be destroyed and replaced by updated agents using Java's dynamic class loading feature. Furthermore, the architectures based on mobile agents are a lot easier to redesign as compared to traditional client/server models.

7. This scheme also works in unreliable network conditions since agents can be transferred using UDP.

8. Since mobile agents are a push technology, they provide increased ***flexibility***. This research regards flexibility as the biggest advantage of this paradigm.

9. An agent operates independently of the application from which the agent was invoked. This autonomy provides the advantage of asynchronous communication. This feature is very useful for the applications/users that either want to work asynchronously or offer disconnected operations (e.g., mobile users).

10. Developing (programming) agents is more straightforward than standalone application development because a mobile agent system usually provides a generic agent structure that can easily be tailored to the requirements of an application.

11. Mobile agents are a good choice for any client/server application in which a server interacts with many clients or there are many interactions between two parties to accomplish a task. By sending an agent to handle the query or interaction, network bandwidth consumption is reduced because the agent performs much of its processing at the server side where local bandwidth is high. The agent communicates locally to the client and instead of sending intermediate results and information over the network, only the agent is sent. Since this PnP application requires queries and negotiations with several client components, therefore, the use of mobile agents (discovery agent, insertion agent, upgrade query agent) lowers the network bandwidth consumption. Note that, an equivalent CORBA object must make repeated remote method calls to the destination object to achieve the same task, causing a lot more

traffic flow[4] and thus more bandwidth consumption.

12. Mobile agents are good choice for the scenarios when a server wants to collect data from many sources. For example, a PnP component needs to collect data from other available client components in this PnP application. Instead of querying the components separately, the PnP component sends an agent (SRD-Agent) over the network with a travel plan and the agent navigates through all these components and collects data. This way the PnP components may also work asynchronously after dispatching the agent.
13. Filtering relevant information from the vast amount of data can be a very frustrating and time consuming process. A mobile agent can perform this task on behalf of the user and thus provides another important advantage. In this application, the discovery agent navigates the entire network but retrieves just the relevant information (possibly, from a large data set) from the interested components and presents this filtered information to the PnP component.

5.4.2 Disadvantages

1. For running this mobile agent based PnP application, an agent system (MCT) must be installed on each network component. Moreover, in order to execute an agent system, the components must be language-enabled (i.e., Java-enabled in our case). These

---

[4] assuming calls to be moderately heavyweight.

requirements incur large overheads as they consume significant computational resources (CPU, memory).

2. Security is the biggest issue associated with any code that is sent across the network. Since a mobile agent may traverse many network components, security is even a bigger concern. Therefore, well-defined and powerful security techniques must be deployed to ensure security.
3. Permanent discovery agents are always alive so they consume considerable network bandwidth and may lower the throughput of the network if no attempt is made to control the density of such agents.
4. The lack of standardization and interoperability (despite the current efforts like MASIF) are among the major issues that are impeding the growth of mobile agent technology.
5. Since mobile agents have their origins in the AI world, therefore they inherit many drawbacks and issues associated with typical agent-oriented projects, as illustrated in [52].

This evaluation section ends with a quotation from a comprehensive research report on mobile agents [4]. This report discusses different applications of mobile agents and suggests alternative client/server solutions. The report concludes with the following:

> *“ … While none of the individual advantages of mobile agents …is overwhelmingly strong, we believe that the aggregate advantages of mobile agents is overwhelmingly strong because:*
>
> *a. They can provide a pervasive, open, generalized framework for the development and personalization of network services.*
>
> *b. While alternative to mobile agents can be advanced for each of the individual advantages, there is no single alternative to all of the functionality supported by a mobile agent framework.”*

Having experienced a mobile agent based development through this PnP application, the author agrees fully with the above conclusions.

Chapter 6

Jini-based PnP Network Components

Jini is a Java-centric distributed computing technology that facilitates the concept of plug-and-play by providing the necessary networking infrastructure and distributed programming facilities. Jini allows us to create a *federation* (community) of hardware and software components in a single distributed space on the network, where the components may dynamically connect to share services without any prior knowledge of the existing network environment. The fundamental unit of a federation is a service and any Jini-enabled device or user application can function as a "service" on a network.

The greatest promise offered by Jini's service-based architecture is *spontaneous networking*. This means that the addition of a service to the network or plugging in a new device would expose the service immediately to all potential clients. A client can then use the service without having any prior knowledge of the service location and implementation. A Jini-enabled printer, for example, may offer a "print service" to all the other members of the federation as soon as it is connected to the network. The enormous potential of Jini technology for plug-and-play environments makes it an ideal enabling technology for implementing the proposed PnP approach for network components.

This chapter describes the Jini-based PnP approach for configuration management of network components through a detailed design and an implementation. This chapter is composed of two main parts – the first part provides an overview of Jini technology and its architecture and the second part describes the Jini-based design and implementation. This chapter concludes with a description of the pros and cons of the technology.

6.1 JINI OVERVIEW

Jini technology [42] is originated from an R&D project that was first conceived in 1994 at Sun Microsystems. The project achieved its first two milestones -- developing a network programming language and a distributed computing technology -- with the introduction of Java and RMI respectively. The stage was then set to achieve the final goal of developing an infrastructure that would allow construction of a federation of services over a network with minimum efforts. Jini is the technology, which ultimately achieved this goal.

Jini is based on RMI technology [37] and is designed to extend the Java application environment from a single virtual machine (VM) to a network of machines. Services in a Jini system communicate with each other through a set of *service protocols* and a *service brokering* mechanism, known as the Lookup service (LUS) [47]. The service interfaces specify functionality provided by a service and the LUS maps these interfaces to the set of objects that actually implement the service. Services advertise their functionality in the LUS using the *discovery* and *join* protocols and clients find and resolve their required services through the LUS using the *discovery* and *lookup* protocol.

Jini technology uses RMI's security and code mobility features. RMI also allows for communication between different objects and provides the mechanisms for finding, activating, and garbage collecting the objects (or the object groups).

6.1.1 Jini's Applications

Jini technology can be applied to a wide variety of areas. Typical areas include consumer electronics, computer peripherals, enterprise computing, home environment, electronic commerce, and several others. Furthermore, the promise of Jini has captured the attention of researchers in various areas such as programmable networks [7], telecommunication service provisioning and intelligent networks.

6.1.2 Jini's Competitors

Lucent with InfernoSpaces [21], Microsoft with 'Universal Plug-and-Play' ([25], [27]), and Hewlett Packard with ChaiAppliance [14] all are in race with Sun Microsystems' Jini to develop the next generation of distributed computing technology that would allow devices -- from a toaster to a supercomputer -- to be connected in a ubiquitous network. Although industry analysts consider Jini and Universal Plug-and-Play to be direct competitors, Jini has a definite edge due to its open source availability and deployment in several consumer devices at this moment.

6.2 JINI ARCHITECTURE

The Jini architecture [39], as shown in Figure 6.1, can be segmented into the following three components:

1. runtime infrastructure
2. programming model
3. services

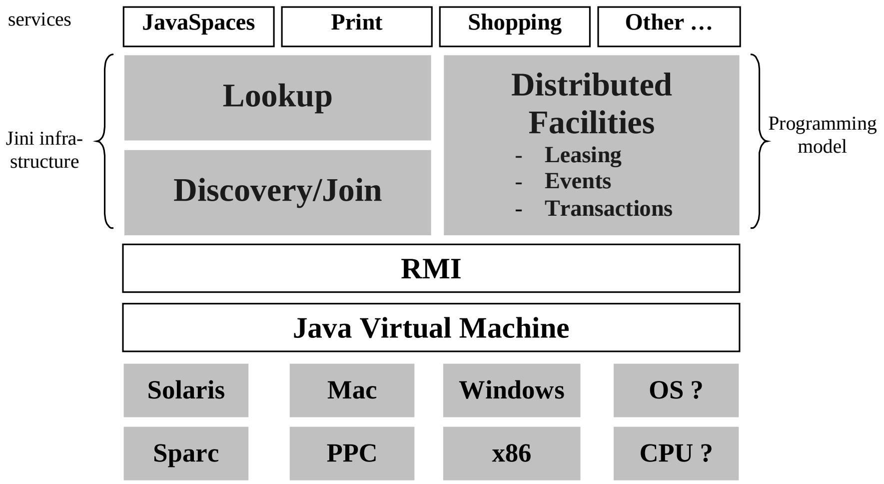


Figure 6.1: Jini system Architecture

6.2.1 Runtime Infrastructure

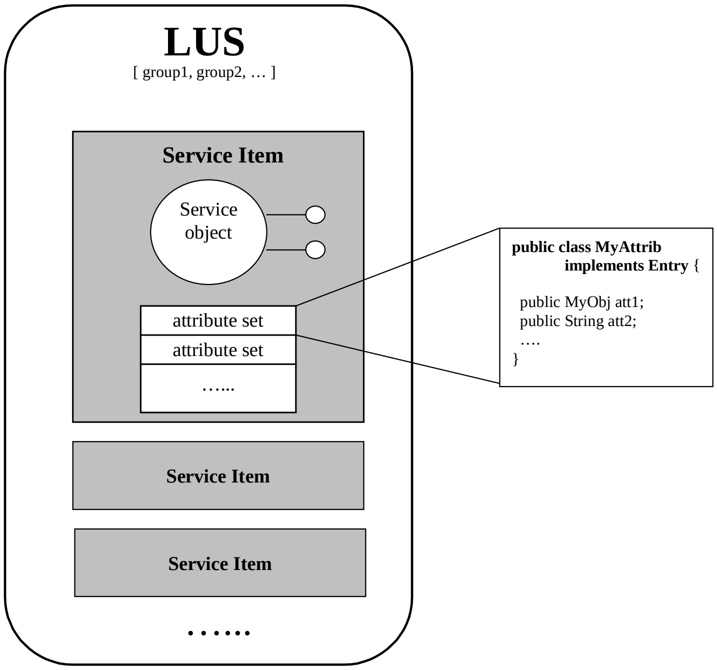
The Jini infrastructure consists of a set of components that allow construction of a federated Jini system. The runtime infrastructure is the core Jini software that provides mechanisms for adding, withdrawing, locating and accessing services on the network. Services use this infrastructure to advertise themselves when they join the network, whereas clients use this infrastructure to locate and contact services. The infrastructure is composed of the following components and features:

1. Lookup service
2. Discovery, Join, and Lookup protocols
3. Distributed security feature
4. Code mobility feature

6.2.1.1 Lookup Service

The Lookup service [47] is the fundamental component of the Jini architecture. When a new device is plugged into the network (or a software service is started), it registers with a LUS to join the federation. Clients may consult the LUS to discover their required services. In other words, a LUS is a *bridge* between services and clients and acts as a *service broker* component.

Services are segmented into logical groups such as “admin”, “printers”, “finance-dept”, or “main-room”. Consequently, each LUS is associated with a logical group(s) and is responsible for registering only a particular group of services. For example, all the interested printers join a LUS that represents the “printers” group. It is to be noted that a LUS may represent many groups and a service may also be a part of many groups. Moreover, a group can be maintained in multiple LUSs. Although redundancy raises the issue of synchronization, it is recommended here for fault-tolerance reasons. A LUS may



```
public class MyAttrib
        implements Entry {

  public MyObj att1;
  public String att2;
  ….
}
```

Figure 6.2: Jini Lookup Service

also contain a pointer to another LUS, thus providing a hierarchical and scalable lookup facility.

Figure 6.2 illustrates a high-level view of a typical LUS, where a central registry for services is provided by maintaining a flat list of *service items*. A service item consists of two entities:

1. *service object*: an instance of service code corresponding to a service available in the federation.
2. *service attribute sets*: descriptive entries that allow for a more fine-grained selection of services.

The runtime infrastructure enables services and their clients to access and use the LUS through the discovery, join, and lookup process.

6.2.1.2 Discovery process

Discovery is the process of locating the existing Jini federation or community [44]. A service component may want to discover a LUS (s) to advertise its service, whereas a client may want to discover the federation to locate a desired service. Jini technology relies on unicast and multicast IP facilities and supports three different discovery modes:

1. A LUS announces its presence on start-up.
2. Clients/services may initiate discovery for a nearby LUS (within the multicast range).
3. Clients/services may use unicast discovery to communicate with a *specific* known LUS, which is situated beyond the multicast range.

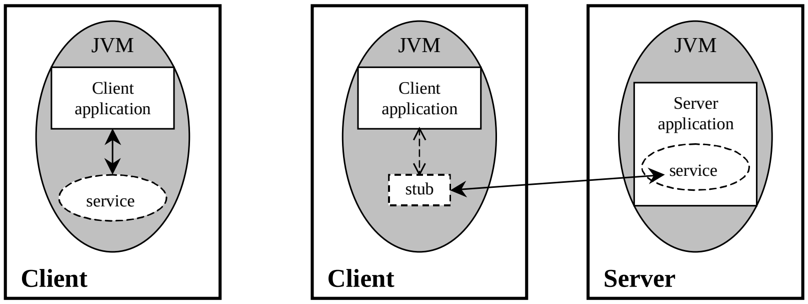
As a result of a successful discovery, the LUS contacts and uploads its *service object* (LUS-Proxy) to the discovering entity (client/service). The proxy object of a LUS implements certain interfaces (*register, lookup*, and *notify*) that the clients/services may use later to interact with the LUS.

6.2.1.3 Join process

Once a device/service has discovered the required LUS, it can register or advertise its services with that LUS using the *join* protocol. The service constructs a service item that consists of a service proxy object and several associated service attributes. The service advertises this service item by invoking the *register* interface on the proxy object of the LUS. Every service is assigned a *service ID* when it first starts up. This ID is unique across time and space and must be maintained by the service for further registrations or negotiations. The LUS assigns a service ID to the registering service, stores the service item, and associates the item with the corresponding group(s).

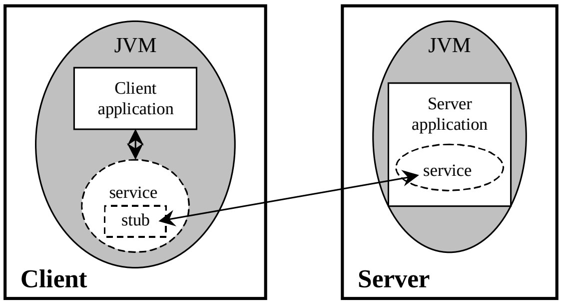
When a client performs a *lookup*, it is the service object (proxy) part of the service item that actually gets downloaded to the client from a LUS. The service object implements certain *service interfaces* that are used by a LUS to identify and match a service to a client's requirement at the time of lookup. On the other hand, a client uses these interfaces to interact with the actual service after a successful lookup. Jini provides us freedom in structuring this service object. As shown in Figure 6.3, there are three different possibilities for implementing a service:

1. **Local service**: all the interactions between a service proxy object and a client occur within the client's VM.



(a) Local service is downloaded from LUS, instantiated and executed in client’s VM.

(b) remote stub is downloaded from LUS that communicates with the remote service at server ’s VM.



(c) smart proxy is downloaded from LUS, instantiated and executed at client’s VM. It communicates also with the remote service part at server’s VM

Figure 6.3: Service implementation modes

2. **Remote service**: The service proxy object is the RMI stub of the remote service. Any call that the client makes to the service proxy are actually handled by the stub (using standard RMI) and executed on the server's VM with the results transferred back to the client’s VM.

3. **Smart proxy service**: Sometimes a service may want some actions to be executed locally on the client’s VM and others to be performed remotely on the server side. In such cases, the service object contains both the local service code and the stub of the remote service.

As mentioned earlier, a service item also includes optional service attribute objects to describe the service further. These attributes may define anything related to the service --- service information, description, status, and user interfaces.

6.2.1.4 Lookup process

Once a service joins at least one group, it is available within the federation and may be found by clients via the *lookup* process.

Having discovered a LUS through the discovery process, a client invokes the *lookup* interface on the lookup server and requests a service of a particular type. The client specifies its requirement using a *service template*. A service template specifies the *type* of service required (in terms of Java classes/interfaces) and a set of *attributes*. A client may also specify the service ID (if known in advance) to locate a particular service. All these fields (i.e., type, attributes, and id) are optional and may not be specified by a client for matching. For example, a service template that specifies wild cards for all of the fields is a valid template and results in the matching of all the services registered in the LUS. As a design rule, however, a client should locate a service by specifying its:

1. type, and
2. attributes (to qualify several potential matching services).

The LUS tries to match the specified template with each stored service item using Jini's LUS matching mechanism [47], which also supports the use of wildcards. As a result of the lookup request, the LUS returns *zero to many* service objects (proxies) to the client that have correctly matched the service template.

6.2.1.5 Client/service interaction

After a successful lookup, the client has the proxy object of the service downloaded from the LUS. The client can interact with the service through *well-known* service interfaces. At the same time, a client can also use Java's *reflection* APIs to look for other interesting methods declared by that object. Having found an interesting method that it understands, a client may interact *dynamically* with the service using this method. A service object may itself be an applet that provides the interaction interfaces to a human user via a simple graphical user interface.

A service object can either be a local, remote, or a smart proxy implementation of the service, as described earlier. The remote/smart proxy implementation of the service object may use a *private protocol* for communicating transparently with its remote service. For example, a proxy could communicate with its service via CORBA, DCOM, RMI, a mobile agent, or some proprietary protocol.

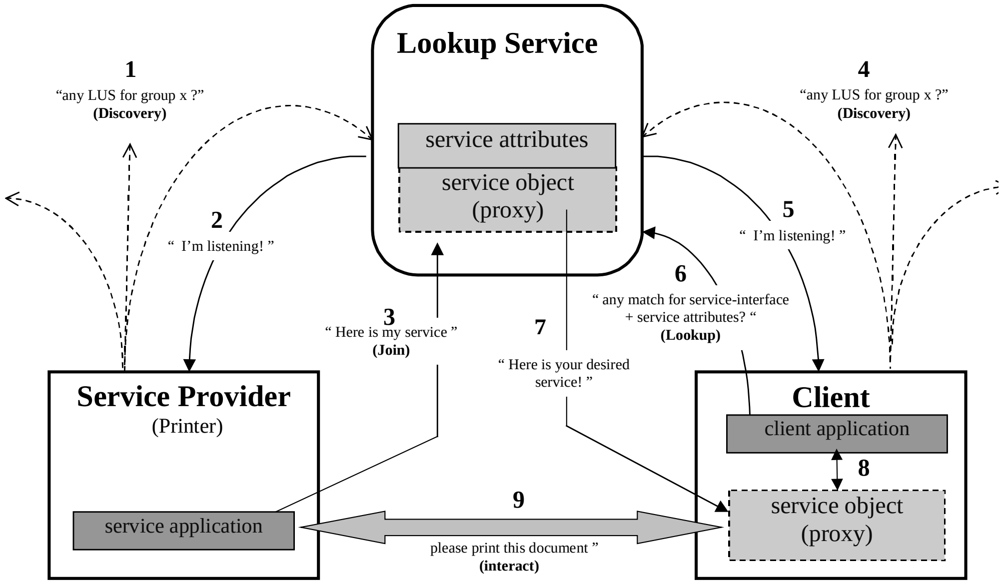


Figure 6.4: Jini system functionality

Figure 6.4 presents the high-level functionality of the Jini-based system by illustrating the discovery, join, and lookup processes for a printer device.

6.2.1.6 Notification facility

The proxy object of a LUS also defines an interface method (*notify*) that provides the *notification* feature through an event registration mechanism. For example, a client who has not found its required service in a LUS may register for event notification from the LUS should such a service arrive in the future. To do this, the client defines a service template, passes the reference of its remote event listener, and specifies the conditions for event notification. Jini defines three conditions on which a LUS may generate a notification. These conditions depend on the matching status of the service template at event registration time and at event generation time.

This PnP application takes advantage of Jini's notification mechanism and demonstrates that it is an important feature for enabling plug-and-play environments.

6.2.1.7 Distributed Security

Distributed security is an important component of the Jini runtime infrastructure. Jini technology uses the built-in security of RMI, which extends the Java platform security model to distributed systems. The Jini security model suggests the use of a *principal* and an *access control list*. That is, the ability to access a service depends on the contents of an access control list that is associated with the caller object (principal). In the current release of Jini, a minimal level of security is provided and security mechanisms are left up to the application developers.

6.2.1.8 Code mobility and codebase

The Jini technology uses the *code mobility* feature of RMI for passing the service code to a client. The service code is sent using RMI in the form of a *marshaled object* that is un-marshalled at the receiver's end. During the process of marshalling, a sender serializes the object and annotates the location (URL) of its codebase to tell the receiver where the sender's class files are located and how they should be retrieved. The receiver extracts this annotated codebase during the un-marshalling process and uses it to download the required classes. This mechanism, however, requires a class server (e.g., a simple HTTP server) that will allow the sender's classes to be transmitted over the network.

6.2.2 Programming Model

The Jini programming model consists of a set of application programming interfaces defined for developing Jini services and other distributed systems. The Jini infrastructure makes use of these standard programming interfaces to construct a federation of services. The participant applications may also use these interfaces to perform their application tasks.

The Jini programming model is composed of the following interfaces:

1. leasing
2. event and notification
3. transaction

Distributed *leasing* is a time-based model for activating and freeing services and resources. A lease is a grant of guaranteed access to a remote service/resource for a specified period of time [46]. The participants in a Jini system negotiate the leases with each other to insure the availability of a service/resource during the leased period. Furthermore, leasing provides a way to manage the lifetime of distributed objects, thus facilitating distributed garbage collection. A lease can be renewed by a leaseholder before its expiry period. Note that service registrations with a LUS are lease based, as are the registrations for event notification.

Jini's distributed *event* model [45] extends the JavaBeans event model and provides a notification mechanism between distributed services and resources. Using this model, a client can register its interest in some events with a remote source and the client is notified when that remote event occurs. The event facilities in the Jini programming model are very important as they enable development of *dynamic* and event-driven distributed applications.

Distributed *transactions* [48] allow a series of operations, spread across multiple services, to be grouped into a single unit of work. The transaction model uses a two-phase commit protocol between participants, resulting in a reliable distributed system. The transaction facility is important for business and database applications.

6.2.3 Services

The fundamental concept in the Jini architecture is a "service". A service is an entity within a federation that can be used by a person, program, or another service. From Jini's point of view, *everything is a service* -- a computation, storage, printing, display, database retrieval, or even a file. Programmatically, the services are ***Java objects*** with particular interfaces that are used by clients to find and access the service. Jini services are mobile since the code is first uploaded from its source machine to a LUS and then downloaded from the LUS to a client machine.

Jini provides two built-in services -- JavaSpaces and Transaction Manager. JavaSpaces is itself a technology, which provides a mechanism for sharing, coordination and communication of distributed objects [38], whereas the Transaction Manager facilitates transactions through a two-phase commit protocol. Jini services, whether initially supplied with Jini or written by the end programmer, make heavy use of both the Jini infrastructure and the programming model. Note that although the word "service" is also used for a LUS, however, a "lookup service" is not strictly a Jini service but an integral part of the Jini's infrastructure.

6.3 JINI SYSTEM ASSUMPTIONS AND REQUIREMENTS

6.3.1 Environmental assumptions

There are certain fundamental environmental assumptions that one needs to make for running Jini:

1. Network of reasonable speed (10 Mbps).
2. Latency is at most in seconds[1].
3. Each connected network device has some memory and processing power.

---

[1] This latency not only refers to a network but is also associated with discovery protocols.

4. Java is the implementation language for Jini components.

6.3.2 Software requirements

The following software technologies are required to run a Jini based distributed system:

1. Jini software version 1.0.
2. Java virtual machine (JVM) and Java Development Kit (JDK) version 1.2 .
3. RMI activation daemon: It provides a persistence facility and is required by the system for recovering after a crash. If a system crash occurs, the services that were registered with this daemon will restart automatically with the daemon restart and will recover their last saved state prior to the crash. Furthermore, the daemon also provides the important property of activating/de-activating services on demand.
4. Network class server: Since Jini's service architecture depends on code downloading from a code-base, therefore, a class server is required to serve as a code-base (a file system or an http server).

6.3.3 Device requirements

The aforementioned assumptions and technology requirements raise certain requirements for a Jini-enabled component:

1. A functioning JVM with JDK, Jini and RMI packages.
2. A properly configured network protocol stack (currently only IP is supported):

  - device needs an initial IP address;
  - protocol should support multicast IP facility.

3. Hardware requirements:
  - Microprocessor: to run JVM, Jini and PnP software;
  - Memory: for creating and loading objects;
  - Storage (disk/NVRAM): to store JVM, Jini, and JDK software.

The Jini specification [43] specifies three possible implementations for the Jini device architecture:

1. A device with a resident JVM.
2. A bay device having cluster of devices and a *single* physically-shared JVM.
3. Device cluster with a *separate* shared JVM (physical) on network.

Since each of the above configurations has some advantages and disadvantages, therefore, it is assumed that every network component in this PnP scheme has a resident JVM.

6.4 PNP NETWORK COMPONENTS: THE JINI DESIGN

The following sections present a design of a Jini-based approach for PnP network components by using an example of a network printer. The Jini-based PnP approach takes advantage of the Jini technology infrastructure and its service brokering architecture for discovering the network and for arranging service modules. The approach does not necessarily require a Jini infrastructure once a client receives the actual service module and thus falls back to the dedicated protocols; i.e., a service module (printer driver in this example) may now use its proprietary protocol to talk to the actual service (printer).

6.4.1 Scheme Layout

Before describing the Jini-based scheme, a high-level view of the Jini-based PnP scheme is shown in Figure 6.5. The diagram illustrates the participant components and other building blocks. The components shown in the diagram are Jini-enabled and conform to the requirements put forward in section 6.3. They implement appropriate discovery/join/lookup protocols, discover each other through LUSs, and interact using service code. It is assumed that an administrator supplies the initial configuration information (and security related configuration) to the components before installing them. These components are distributed over two segments -- an enterprise segment and a

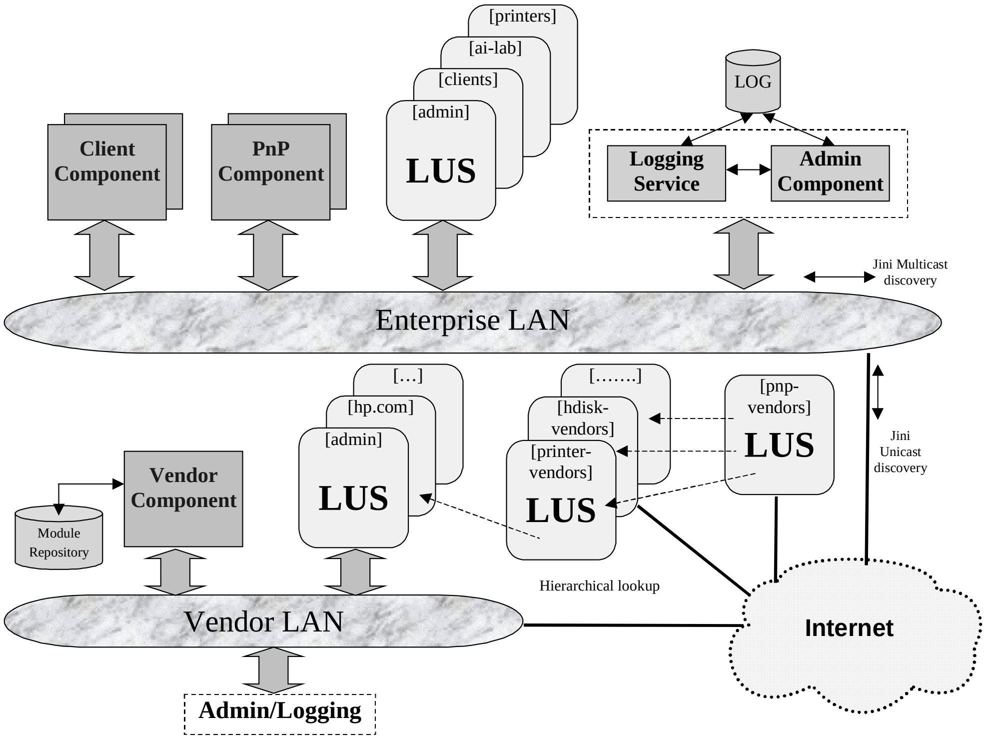


Figure 6.5: Scheme layout

vendor segment.

First, we consider the enterprise network that runs many network components and services. The components connected to the enterprise's LAN are PnP network components, client network components, admin component, logging server, and several lookup servers. These lookup servers represent various groups of services available on the network, such as "admin", "managed-clients", "printers", and "ai-lab". Although, a single LUS may store all the services in a small network, the use of multiple LUSs is mandatory for larger networks. The use of multiple LUSs not only helps in partitioning the services; it also increases *scalability* and *fault-tolerance*.

The second segment of the diagram illustrates the vendor side of the network. Each vendor maintains its network on which many Jini-enabled components are running. The vendor's LAN is connected to the Internet, which allows communication between the vendor and the components on the enterprise network. The central component for PnP negotiations on a typical vendor site is the vendor component. Other components are also running on the vendor's LAN, including the admin component, logging server, and several lookup servers.

There are at least two LUSs available on the vendor's LAN, one each for the admin and the vendor component. The LUSs from different vendors must use standard distinct/unique names. One may obtain a unique name through some standard body but this scheme suggests the use of fully qualified DNS-style names for vendor LUSs -- Hewlett Packard runs a LUS for a group called 'hp.com', whereas IBM maintains its LUS for a group 'ibm.com'. The use of DNS style names solves the issues of uniqueness and standardization without any extra effort on our part.

It is to be noted that Jini-enabled components use Jini's multicast discovery protocol for discovering the LUSs within the LAN boundary, whereas Jini's point-to-point unicast discovery protocol is used to discover and access a LUS that lies across the Internet.

PnP Component

In the context of this thesis, a Plug-and-play (PnP) component is a hardware device (network printers, network hard disks, etc.) that provides service to other network components[2]. The component has a pre-installed server-side module and the hard-coded information about its vendor and vendor's LUS. Each service provider PnP component registers with an appropriate LUS on the enterprise LAN; e.g., a PnP network printer registers with a 'printers-LUS', while a PnP network hard disk registers with a 'hdisks-LUS'. The registering component advertises its offered service by placing its service proxy object in the designated LUS. This proxy provides one of the following facilities:

1. provides service directly to the client, or
2. acts as an intermediate service code that would arrange downloading of the actual service code (e.g., arranging a printer driver for the client).

Client Component

The network components, such as end-hosts, that require access to network services are called the client components. In addition to common configuration information, the

---

[2] A Jini-based PnP software service can also be easily implemented.

administrator initially specifies the component's service requirement and upgrade interest. Even though a client component does not offer a service, it is recommended that client components register with a 'clients-LUS'. A client component, thus, stores its service proxy object in the LUS that helps a network administrator in configuring and managing the component through its service proxy.

The clients find their required PnP components (services) through the designated LUSs. The service proxy of a PnP component gets downloaded to the client's machine that is used by the client for interacting with the PnP component.

Admin component

This component represents an administrator and is responsible for installing, stopping, resuming, killing, and configuring a service (or component). The participant components use this component to send various messages and events. These events and messages are displayed on a console area and help an administrator in monitoring the system-related activities. The logging service is a helper component on the network that collects the messages and events from various network components and logs them appropriately. The admin component interacts with this logging service to check and retrieve the logged messages. Both the admin and the logging components register with the 'admin-LUS' and store their service proxies. These service proxies are downloaded by other network components and are used for communicating admin and logging related activities.

Vendor Component

A vendor component represents a vendor application in this PnP scheme. Since a vendor may manufacture and supply various hardware devices (PnP components), therefore he is required to maintain a repository of the service (device) modules for each component. The vendor registers with its LUS (e.g., ‘hp.com-LUS’) and uploads its service proxy object to it. The advertised vendor proxy provides the interfaces that are used by a PnP component in registering with the vendor. The vendor also stores service module proxies, corresponding to many platforms for various products, in its LUS. The interested clients and server (PnP) components download these module proxies during the PnP installation. The vendor is also responsible for keeping its repository of modules and the corresponding LUS items up to date. Placement of the service modules in a LUS facilitates the mechanisms of module lookup, matching, and downloading by the clients. Furthermore, the vendor LUS may send the module upgrade notifications to the interested clients without any direct involvement of the vendor. This mechanism, however, raises a security issue and requires the service module to authenticate the client component appropriately.

Since the vendor’s LUS is running on the vendor network and the vendor’s PnP component is installed on an enterprise network, therefore the PnP component must use the Jini’s unicast discovery protocol to contact the vendor LUS. This constraint requires a vendor to hardcode the name and address of its LUS in the PnP component at manufacturing time.

Service object or service proxy

The service code that is stored in a LUS by a registering entity may not *directly* provide a network service and may only be used to make *connection* with its source component. For example, a PnP printer stores its service object in a LUS that helps a client in accessing and using the network print service. On the other hand, a vendor also stores a service object in its LUS that does not provide any service and is used only to help its components in interacting with the vendor. Since Jini uses the terms ‘service object’ and ‘service proxy’ to refer to any code that is stored by a component in a LUS, therefore, this chapter also conforms to the same terminology. Although it is only the proxy object of a PnP component that provides an access to a real network service, the term ‘service proxy’ is also used for service objects of client components, vendor component, admin component, and logging service.

6.4.1.1 Hierarchical lookup

The requirement of hard coding the vendor’s LUS information in a PnP component creates certain problems. Consider relocation of a vendor LUS from one host to another. All the PnP components that were manufactured earlier have out-of-date information now and are unable to access the vendor. The use of a hierarchical lookup facility is proposed as a solution to this problem; e.g., a printer vendor’s LUS (‘hp.com-LUS’) registers with a centralized LUS called the ‘printer-vendors-LUS’. The printer vendor LUS itself is

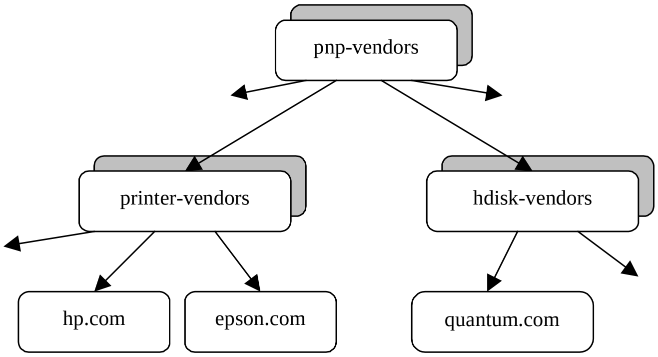


Figure 6.6: Hierarchical lookup

registered with the top-level centralized ‘pnp-vendors-LUS’. If the vendor’s LUS or its parent LUS (i.e., printer-vendors LUS) changes its location, the higher level LUS is kept informed about this relocation. As long as the location of the top-level LUS is not changed, the PnP component may still find and communicate with a relocated vendor LUS. However, in this case, the component needs the hard-coded information regarding the top-level LUS (i.e., pnpvendors-LUS). The use of hierarchical structure also allows the *scalability* and *flexibility* in the deployment of LUSs. Again, it is recommended to have *multiple* centralized LUSs for making the system more fault-tolerant. Figure 6.6 shows a typical *hierarchical* structure for LUSs.

6.4.2 Services and Interface standardization

One of the requirements of Jini’s service architecture is that the clients must have some prior knowledge of a service and its *interfaces*, i.e., although a client need not understand how a service is implemented, it must know how to interact with the service.

This PnP scheme assumes that the clients have a priori knowledge of at least one *“well-known”* Java interface that a service object implements. Although, no well-known interfaces exist for Jini-based services currently, the emergence of standard interfaces from specialized industries (that have the knowledge for defining these interfaces) is expected. For example, the various vendors of printers may get together to define a hierarchy of interfaces for printing. A standard high-level interface for printers (say “*BasicPrinterInterface*”) can be defined that all vendors may extend by defining vendor-specific extensions; i.e., Hewlett Packard could define separate interfaces for color extensions of its printers as *HPColorExtensions* and would make all the HP printers implement both the *BasicPrinterInterface* and its extensions. A client may now use the high-level printer interface or use the extensions implemented by a specific vendor.

From the service provisioning point of view, two kinds of interfaces (Figure 6.7) are proposed that must be supported by a proxy of a registered service:

1. Interaction interface
2. Management interface

The clients use the *Interaction* interfaces to interact with the service, whereas the management interface helps an administrator in configuring/managing a service/device.

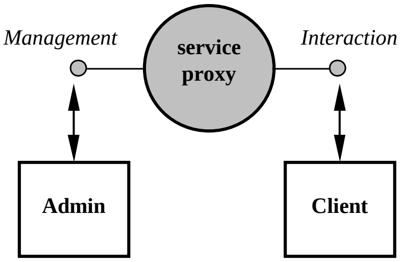


Figure 6.7: service interfaces

6.4.2.1 User Interfaces

Clients need not have a priori knowledge about a service if the service offers a user interface (UI) that a client can display. The client may use this UI to interact with the service without knowing what interface methods are being called. Such service UIs are beneficial to the services whose end-clients are human. For example, an administrator may manage/configure a service through a nice, easy-to-use UI, which is supplied by the service proxy of a component.

From a design point of view, it is not a good idea to make the service object itself a UI class (applet, panel, or frame) and a better way of incorporating UIs into the service is to associate a UI as an attribute of the service. One of the reasons to keep a UI out of the service object is that the service clients may not always have UI classes or display capability available and thus a service that itself is UI-only would not let those clients interact with the service. Therefore, each service in this PnP application has two sets of interfaces as shown in Figure 6.8:

1. Direct programmatic interfaces
2. UI interfaces

A client that does not support a UI can use the programmatic interfaces, whereas any client that is capable of communicating through a UI may use these UI interfaces.

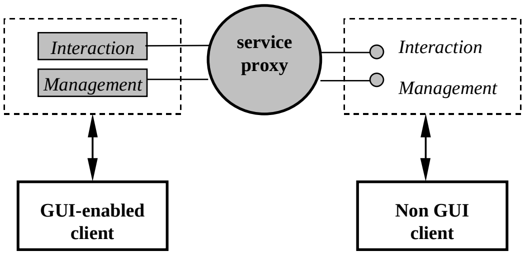


Figure 6.8: User interfaces

There are certain implementation issues associated with the incorporation of UI elements in the services. One of the issues relates to the serialization problem with the Java's UI objects, while the other issue concerns the UI initialization. For example, a UI element may require certain information (such as security) from its user before allowing it to interact. Therefore, it is suggested that each service UI has an initialization method, say *init()*, that a GUI user invokes first for passing the necessary information required by the UI for further interactions.

6.4.3 High-level Scenario

Figure 6.9 presents a high-level view of the activities of this Jini-based PnP scheme. All the components first advertise and store their proxy objects in their respective LUSs using the mechanisms described earlier. Furthermore, the components also try to find a logging server through the discovery and lookup process. The logging-proxy is

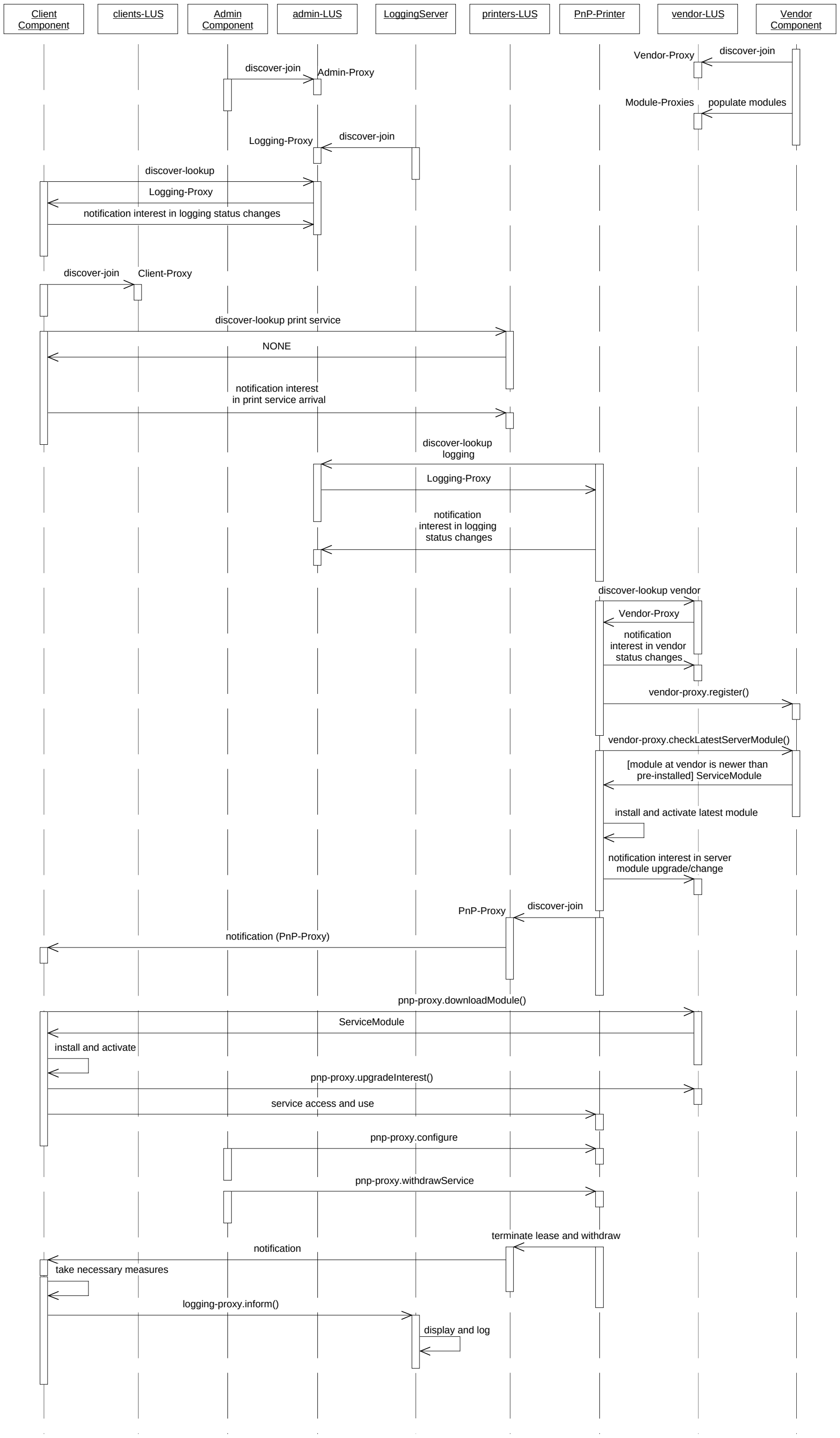


Figure 6.9: Jini-based PnP scenario

downloaded and is used by the components to send any messages/events to the logging service. The components switch to the local logging mode if no logging service is found.

The participant components also register for getting notification of any change in the status of the logging service (e.g., logging service is shutdown or withdrawn). On receipt of such notification, the components switch immediately to the local logging mode.

Now let us say that a client is started that does the discovery of logging and admin components as described above. The client then registers with the ‘client-LUS’ by uploading the Client-proxy. Now, assume that the client wants to use a network printer. The client tries to discover its required service by locating it in the designated ‘printers-LUS’. The client’s lookup fails because there are no printing services available yet. In this case, the client registers with the corresponding LUSs for notifications of the arrival of such a service.

Later on, a plug-and-play printer (i.e., a PnP component) is installed and started by the administrator. Having discovered the logging service, the PnP printer first tries to contact its vendor through the help of the hard-coded location of the vendor LUS. The PnP printer locates the proxy of the vendor in the LUS, downloads the proxy, and registers its interest with the vendor LUS for any status change related to the vendor proxy. The downloaded vendor proxy is used by the PnP component to communicate with the vendor for registration purposes. After successfully registering with the vendor, the PnP printer checks the vendor side for the latest server-side service module. If a newer version of module is found, it is downloaded and installed at the PnP component, replacing the pre-installed module. The PnP component also registers with the vendor LUS for the notifications of server-side module upgrades.

By now, the PnP printer is ready to join the federation by advertising its offered service. The component first finds the appropriate LUS and registers with it by placing its service object (PnP-Proxy) in it. With this advertisement, the clients that have shown their interest in the arrival of such a printing service are promptly notified by the LUS. The advertised PnP proxy is also downloaded to the client side during the notification and the client uses this proxy for interacting with the actual service (printer). In our case, the PnP-proxy does not directly provide the service; it rather arranges the actual service modules (printer drivers) that are necessary for service use. The downloaded PnP proxy hides the module arrangement details from the client and makes all of the arrangements itself. The proxy knows the vendor LUS location since the proxy originated from a PnP component that has this information in a hard-coded form. Thus, the PnP proxy accesses the vendor LUS and looks up a service module that matches the client's OS and platform. The matched client side service module is downloaded to the client machine in the form of the Service Module-Proxy. The module proxy supports a standard interface that allows a client to actually install the module (driver) on its machine. The client may also use the module proxy to register with the vendor LUS for the module upgrade notifications.

At this point, the client has the client-side version of the service module installed that is sufficient for interacting with the service (printer). In our example of actual printing, a client communicates directly with the network printer using the client-side printer driver and the existing networking infrastructure without the involvement of Jini infrastructure.

6.4.4 PnP Interfaces

The Jini service architecture relies heavily on the notion of service interfaces. The services define particular Jini-based interfaces and the clients make use of these interfaces

```
public interface ComponentInfoInterface {
    public Object getAttribute(String name);
}
```

```
public interface AdminInfoInterface {
    public String getID();
    …….
}
```

for service interactions. Furthermore, several Java interfaces have been proposed and implemented in this PnP approach. Although some interfaces are not defined fully, they are still included in the design for the sake of completeness. Following is given a brief description of important interfaces that are defined for this PnP application.

ComponentInfoInterface

This interface allows components to get information about other components. A vendor, for example, may want to query a registering PnP component. Similarly, a PnP component may want to know about its client components during its interaction with them. For querying the information, a component passes an attribute name and the implementer returns the value associated with the attribute. Examples of component's attributes include component's name, owner, location, ip-address, etc.

AdminInfoInterface

This is the interface that allows a component to get information about an administrator (or admin component). It is required when some management operation is to be performed and the managed component needs to first authenticate whether the user is the privileged administrator.

VendorInfoInterface

Like other network components, a vendor is also required to supply its information

```
public interface VendorInfoInterface {
      public Name getName();
      public Address getAddress();
      public URL getURL();
      …..
}
```

```
public interface PnPSecurityInterface {
      public Object getID();
      public PrivateKey getKey();
      …….
}
```

through an information interface. Since the information about a vendor is limited in comparison to the network components’, separate accessor methods are provided instead of using a generic *getAttribute()* method.

PnPSecurityInterface

Due to the importance of security, the use of a security interface is proposed for enabling secure interactions among PnP participant components. That is, the originator of a sensitive request/operation passes its security interface to the receiver of the request and the receiver uses this interface to authenticate the caller and to control access to the resources. This particular PnP implementation does not incorporate this interface.

```
public interface LoggingInterface {
      public void inform(Message message, boolean toBeLogged);
}
```

LoggingInterface

The proxy object of the logging server must implement this interface in order to serve as a data collector of the events and messages. The PnP participant components invoke this interface on the logging-proxy for sending events and messages to the logging server and the admin component.

```
public interface ServiceModuleInterface {
    public String getType();                    // client or server
    public String getVersion();
    public String getProductID();
    public VendorInfoInterface getVendor();
    public String getOS();                      // WindowsNT, Windows95, Solaris
    public String getOSVersion();               // 4.0, 2.6

    public boolean install(ComponentInfoInterface clientInfo);
}
```

ServiceModuleInterface

The Service module is one of the most important objects in this PnP scheme. It can be a simple service proxy, which provides a service *directly* or it may be a device driver (as is the case in this application) that is installed both at the server (PnP) and the client component sides. Although different vendors may support their PnP component through different implementations of service modules, the module always presents a uniform, consistent access to a client as it implements the *ServiceModuleInterface*. The most important method of a module interface is the *install()* method that is invoked by a client component on the downloaded module proxy for installing the actual executables of the device driver. The services that do not require the installation of a device driver may ignore this method.

VendorInteractionInterface

This is the interface that a vendor proxy must implement. A PnP component invokes the *registration* methods of this interface to register or to de-register with the vendor. A PnP component also uses this interface to check and download the latest server module

```
public interface VendorInteractionInterface {
    public PnPCompRegistration register ( ComponentInfoInterface pnpCompInfo,
                                          PnPSecurityInterface pnpCompSecurity,
                                          Lease lease,
                                         LeaseListener listener );
    public boolean deRegister(long pnpCompRegistrationID);

    public ServiceModuleInterface checkServerModule( ServiceModuleInterface pnpCompModule,
                                                     PnPSecurityInterface pnpCompSecurity);
}
```

```
public interface PnPInteractionInterface {
    public ServiceModuleInterface download( ComponentInfo clientInfo,
                                            PnPSecurityInterface clientCompSecurity );
    public EventRegistration registerUpgradeInterest( ComponentInfo clientInfo,
                                            PnPSecurityInterface clientCompSecurity );
}
```

from the vendor side on its start-up.

PnPInteractionInterface

It is a generic interaction interface that the proxies of all the PnP components must implement. The interface is used by the client components for downloading the client side service module and to register with the vendor LUS for module upgrade notifications.

BasicPrinterInterface

It is a high-level interface, which is proposed for the PnP printers. The interface serves as a concrete example of the PnP interaction interface as it extends the generic behavior of the *PnPInteractionInterface* by providing printer-specific behavior. For example, it defines a useful *print()* method that provides the direct printing facility for all

```
public interface BasicPrinterInterface extends PnPInteractionInterface  {
      public PrinterInfo getPrinterInfo();
      public void print( Document document );
      ……
}
```

those printers that do not need to arrange downloading of service modules (drivers). Similar high level interfaces can also be devised for other PnP components, such as PnP hard disks, scanners, etc.

ManagementInterface

Since management of network components by an administrator is mandatory, therefore a management interface is proposed that must be implemented by the components for enabling an administrator in managing/configuring the component/service/device. Using this interface, the administrator issues SNMP-like GET/SET requests to manage the component. The administrator may also withdraw (or set status of) a service of a component for maintenance purposes through this

```
public interface ManagementInterface
                  extends ConfigurationManagementInterface,
                           FaultManagementInterface,
                           SecurityManagementInterface,
                           PerformanceManagementInterface,
                           AccountingManagementInterface {
      public Object get( Object attribute, PnPSecurityInterface adminSecurityInfo);
      public boolean set( Object attribute, Object value, PnPSecurityInterface adminSecurityInfo);
      public boolean withdrawService(PnPSecurityInterface adminSecurityInfo);
      …..
}
```

management interface.

This management interface itself extends many interfaces, corresponding to the functional areas designated for Network Management. The interfaces for the functional areas are included without giving their definitions because the intent here is to emphasize their need and importance by keeping them in the big picture.

```
public interface ServiceUIInterface {
      public Applet getInteractionGUI( ComponentInfoInterface compInfo,
                                                         PnPSecurityInterface compSecurityInfo);

      public Applet getManagementGUI( AdminInfoInterface adminInfo,
                                                         PnPSecurityInterface adminSecurityInfo);
}
```

ServiceUIInterface

Since services may allow their clients and users to interact visually, therefore, all of the proxy components of this PnP application implement a *ServiceUIInterface*. The users invoke this interface to retrieve the actual service UIs from the downloaded proxy. The service UI interface provides two kinds of UI elements -- an *interaction* UI element and a *management* UI. The UI elements allow users to interact visually, while they internally support the direct programmatic interfaces. A user of an UI element, however, must first pass its information and security interface that would be used by the UI element in the actual method calls.

Remote interfaces

Since several proxy objects in this PnP application are the *smart proxy*

```
public interface RemLoggingInterface
        extends LoggingInterface,
        java.rmi.Remote {
}
```

implementation of a service, therefore, they implement the interfaces that are *local* to the client but internally call their *remote* service components for performing the actual operation. In such cases, the remote service implements a prototype *remote* interface[3], which defines the same methods as in the local interface. For example, clients invoke *LoggingInterface* of the downloaded logging proxy, which in turn invokes a remote interface *RemLoggingInterface* on its remote logging server. Note that these remote interfaces do not add any behavior except converting a local interface to a remote one. The examples of PnP remote interfaces are *RemLoggingInterface*, *RemVendorInteractionInterface,* and *RemManagementInterface*.

6.5 PNP SCHEME: STEP BY STEP

Having defined the high-level scenarios and plug-and-play interfaces, this section describes a Jini-based scheme for plug-and-play network components step-by-step. The scheme uses Jini's service brokering architecture for allowing components to discover each other through the LUS and interact through the service proxy objects.

The following is a detailed summary of the scenarios in this PnP scheme. These

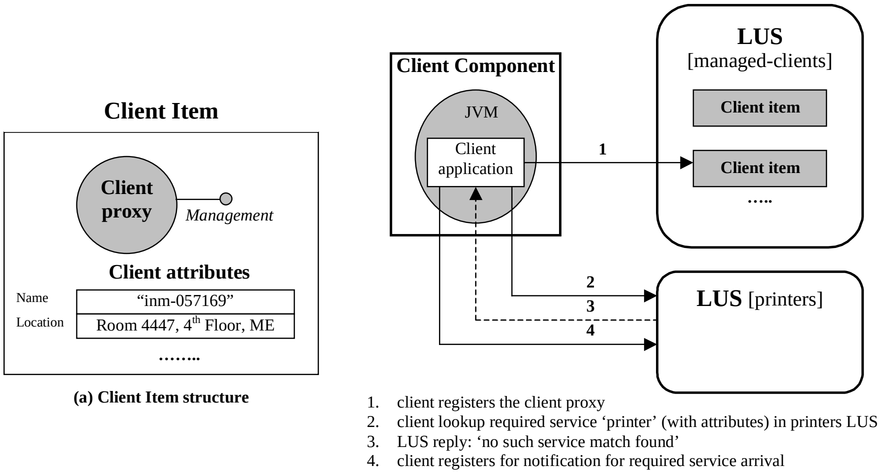
---

[3] java.rmi.Remote

subsections may repeat some of the earlier discussions as they describe the scenarios in considerable detail and depth with the help of diagrams. Although all of the proxy objects in this PnP application implement the *ServiceUIInterface* and enable a user to interact visually, this interface is not shown in diagrams to keep things simpler and more readable.

6.5.1 Client's start-up, registration, and service requirements

When an administrator installs a new network component (such as a PC), he/she first provides a set of initial configuration parameters for the component. The component first discovers the logging server and downloads its proxy for logging and messaging purposes. The client component also has to register with a LUS on the network for management purposes. The component constructs its *client item* and places it in the client LUS. A client item consists of the Client-Proxy and client attribute objects. The client proxy does "not" implement any *interaction* interface and only offers the



(a) Client Item structure

1. client registers the client proxy
2. client lookup required service 'printer' (with attributes) in printers LUS
3. LUS reply: 'no such service match found'
4. client registers for notification for required service arrival

(b) Client registration and service requirement

Figure 6.10: Client component's start-up, registration, and service requirement

ManagementInterface interface so that an administrator may manage the component.

Having registered its proxy in the LUS, the client component tries to find the required service. For example, a client that is looking for a print service with certain attributes first discovers a printers-LUS and then tries to find the matching service in the LUS by specifying the service template. Assuming that no matching print service is available in the LUS, the lookup request fails. The client, in this case, registers with the printers-LUS to receive notification on the arrival of a matching service. Figure 6.10 illustrates the process of client registration and service lookup.

6.5.2 Vendor set up and module advertisement

A vendor of PnP components who decides to support Jini-enabled devices has to maintain an Jini-enabled network as illustrated in section 6.4.1. Having set up the appropriate components, the vendor places its *vendor item* in the vendor-LUS (e.g., hp.com-LUS). The vendor item is comprised of the Vendor-Proxy and the vendor attributes. The vendor proxy implements the *VendorInteractionInterface* and

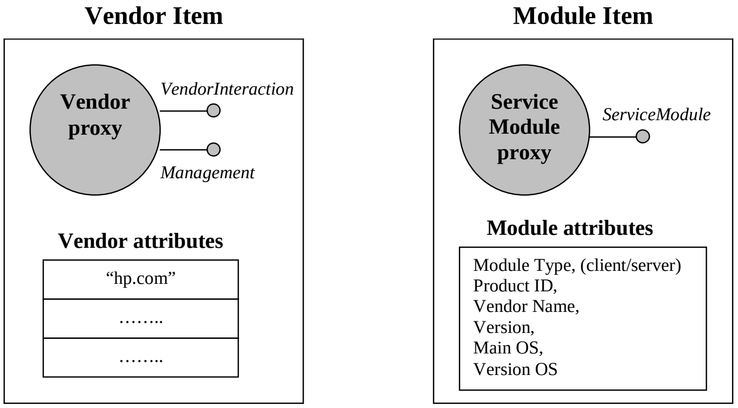
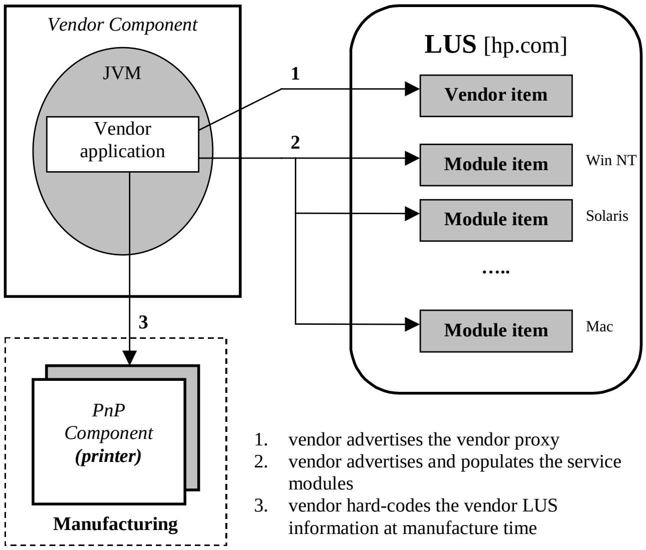


Figure 6.11: Vendor Item and Service Module Item

ManagementInterface interfaces. The vendor proxy is advertised so that the vendor-manufactured PnP components may find and download this proxy for interacting with the vendor. After advertising its proxy, the vendor also advertises the service modules corresponding to the various products and platforms in the vendor-LUS. The advertised *module item* is comprised of the Service Module-Proxy and several module attributes. There are two kinds of service modules stored in the LUS -- a server-side service module that the vendor's PnP components require and a client-side service module that needs to be installed on the clients of a PnP component. The service module proxy implements the *ServiceModuleInterface* that helps a client component in the installation of actual device driver.



1. vendor advertises the vendor proxy
2. vendor advertises and populates the service modules
3. vendor hard-codes the vendor LUS information at manufacture time

Figure 6.12: Vendor setup and module advertisement

Furthermore, the vendor hard-codes the location of the vendor LUS in its PnP components at manufacturing time so that they can interact with the vendor and its LUS after being installed on an Jini-enabled network. Figure 6.12 summarizes the activities and setup of a vendor.

6.5.3 PnP component registration, module downloading, and notification interests

When a network administrator needs a service, say a printer, he purchases an appropriate Jini-enabled PnP component and installs on the network. The new PnP component initializes and discovers appropriate LUSs like any other network component. Before advertising its service, the component tries to register with its vendor. For performing this registration, the PnP component contacts the vendor LUS using its hard-coded information and downloads the vendor's proxy by specifying a matching template. The component first invokes the *registration* interface on the vendor's proxy to register with the vendor. The registration of a PnP component with the vendor is secure and lease-

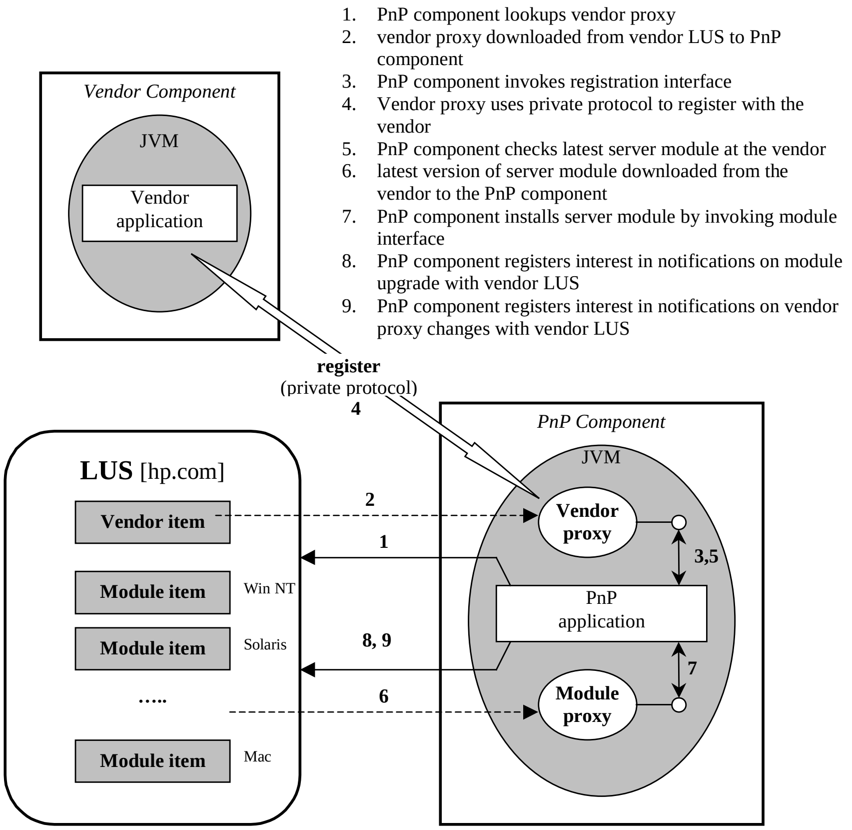

1. PnP component lookups vendor proxy
2. vendor proxy downloaded from vendor LUS to PnP component
3. PnP component invokes registration interface
4. Vendor proxy uses private protocol to register with the vendor
5. PnP component checks latest server module at the vendor
6. latest version of server module downloaded from the vendor to the PnP component
7. PnP component installs server module by invoking module interface
8. PnP component registers interest in notifications on module upgrade with vendor LUS
9. PnP component registers interest in notifications on vendor proxy changes with vendor LUS

Figure 6.13: PnP component registration, module download, upgrade interest

based and a successful registration returns a registration ID (through the *PnPCompRegistration* object) that the component may use to de-register in the future.

After registering, the component checks the server-side module (driver) at the vendor site. If the module is a newer version than the component’s pre-installed module, it is downloaded and installed on the PnP component.

Finally, the PnP component asks the vendor LUS to send a notification whenever the corresponding server-side module is modified or replaced by the vendor. The component also expresses an interest in any changes (e.g., addition of new features, or withdrawal of the vendor) in the vendor status and the vendor LUS notifies the component on module upgrades or changes in the vendor’s status.

6.5.4 Service advertisement by the PnP component

The plug-and-play component discovers the federation and then advertises its service by placing its *PnP item* in the corresponding LUS. The PnP Item consists of the PnP-Proxy and associated PnP service attributes. In addition to the management interface, a PnP proxy implements the service interfaces that correspond to the PnP component’s service. A PnP proxy for a printer device, for example, implements the *BasicPrinterInterface* and *ManagementInterface* interfaces. The *BasicPrinterInterface* is the ***interaction interface*** that defines methods for the

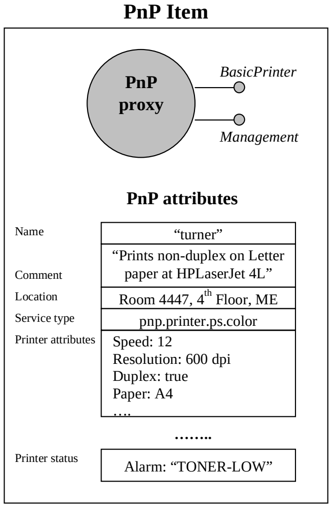


Figure 6.14: PnP Item

actual service module downloading and for registering interest in upgrade notification. Client components invoke this interaction interface when they wish to use the PnP service (i.e., printing), whereas an administrator may use the *ManagementInterface* of the proxy to configure the service/component.

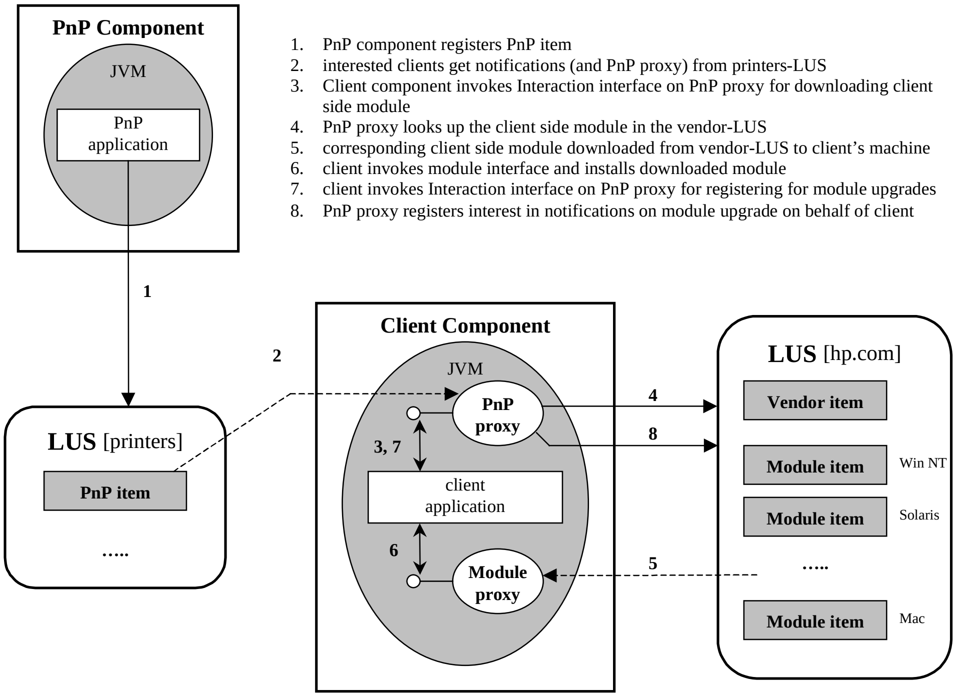
The service attributes are of much importance to an advertised PnP Item. A client uses the service interface (i.e., *BasicPrinterInterface* in this case) to find its required service but reduces the number of potential matching services by specifying certain service attributes. For example, a client asks for a "*printer that may print postscript in color at the speed of 12 pages/minute on A4 paper size*". A PnP printer, therefore, associates its service type (pnp.printer.ps.color) and its attributes (such as speed, paper, and resolution) with the stored PnP proxy as shown in Figure 6.14. It is worth mentioning here that the use of the *service type* attribute conforms to the hierarchical service structure suggested earlier in chapter 3. In addition to common attributes like Name, Location, and Comments in its service item, the attributes may represent the *operational status* of the PnP component. For example, a PnP printer may add an attribute called the *PrinterStatus* to its PnP item in the LUS if it detects some alarm (such as "Toner Low", "PaperJam", "OutOfPaper", etc.), thus causing an interested administrator to be notified through the LUS.

6.5.5 Service arrival notification, client module installation, and upgrade interest

As soon as the PnP item registers with its LUS, the print service becomes part of the federation and any interested client component can find and look up this service. Furthermore, all the clients that have shown interest in such a service (section 6.5.1) receive the notification from the LUS. As part of the notification, the client components also get the actual PnP proxy downloaded to their machine.

A client may now invoke the service interaction interface (i.e., *BasicPrinterInterface*) to download the actual client side service module. The client invokes the *download()* method on the PnP proxy which in turn contacts the vendor LUS to find a service module that matches the client's OS and platform. The matched client-side service module is returned to the client and the client invokes the module interface to install the module (driver) on the client's machine.

Finally, the clients that are interested in receiving module upgrades use the PnP proxy interface to register with the vendor's LUS for getting notifications on any upgrade of the corresponding client module. When the vendor modifies the existing client side module



1. PnP component registers PnP item
2. interested clients get notifications (and PnP proxy) from printers-LUS
3. Client component invokes Interaction interface on PnP proxy for downloading client side module
4. PnP proxy looks up the client side module in the vendor-LUS
5. corresponding client side module downloaded from vendor-LUS to client's machine
6. client invokes module interface and installs downloaded module
7. client invokes Interaction interface on PnP proxy for registering for module upgrades
8. PnP proxy registers interest in notifications on module upgrade on behalf of client

Figure 6.15: Service advertisement, client notification, module installation, upgrade interest

in its LUS, the corresponding clients are immediately notified and the upgraded module is uploaded to the client's machine so it can be installed immediately.

Note that since the client had already registered its interest in the notifications related to the PnP component (printer) changes, therefore it receives a notification message whenever some relevant changes occur in the component status at the printers-LUS (e.g., PnP printer is withdrawn). On receipt of the notification, the clients respond accordingly; i.e., they abandon the use of the service and inform the admin component by sending a message to the logging service.

6.5.6 Service access and use

By now, the PnP component is up and running and the clients are ready to use the offered service since they have already installed the service module (driver) of the PnP component/device. Although Jini has been used to facilitate the entire process of initial configuration, discovery, lookup and module arrangements, it is not mandatory to use Jini for service access and use. Therefore, the PnP application falls back to its proprietary protocol for accessing and utilizing the actual service. In case of our printing service, the

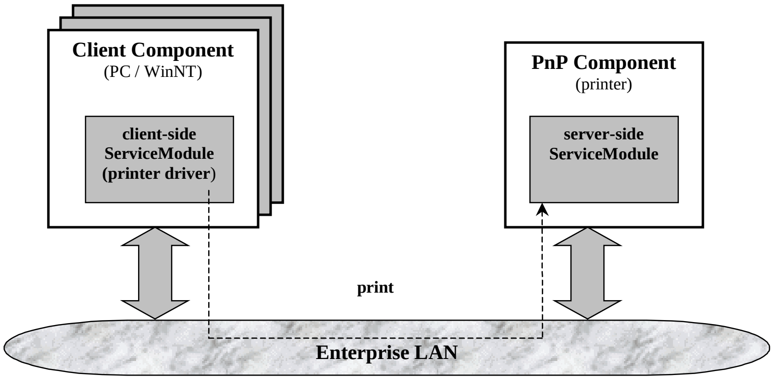
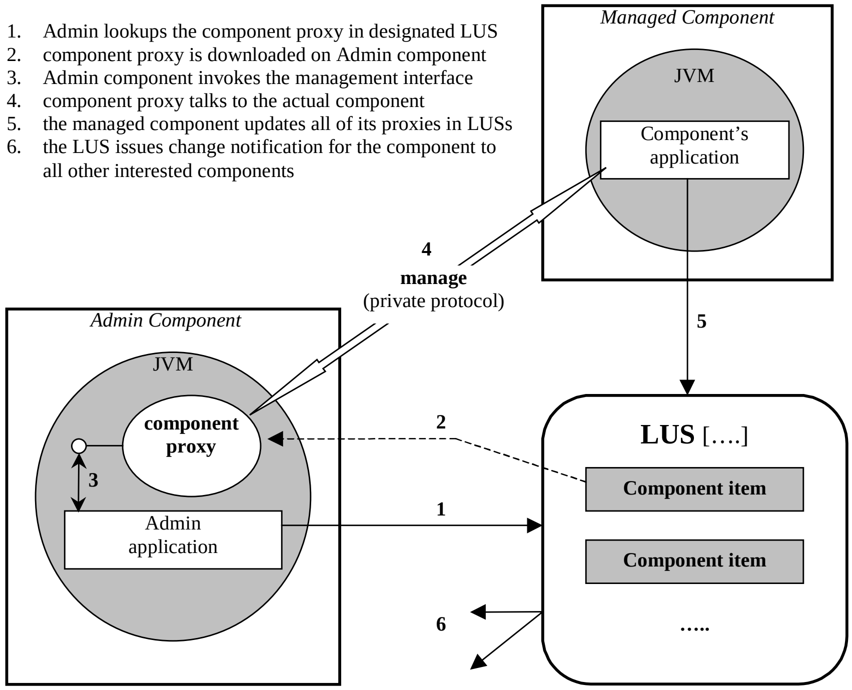


Figure 6.16: Service access and use

client uses the installed client-side printer driver and communicates over the existing network to send print requests to the print server (PnP component) without any involvement of the Jini infrastructure.

6.5.7 Administration and management

Management and configuration of network components by an administrator is an important feature. In a PnP application scenario, there are administrators for managing the enterprise and vendor networks. The enterprise administrator manages the PnP components and other client network components, and the vendor administrator manages vendor components. An administrator who wants to manage/configure a component, first looks up the component in its designated LUS and then downloads the component's proxy. The proxies of the managed components implement the *ManagementInterface*.



1. Admin lookups the component proxy in designated LUS
2. component proxy is downloaded on Admin component
3. Admin component invokes the management interface
4. component proxy talks to the actual component
5. the managed component updates all of its proxies in LUSs
6. the LUS issues change notification for the component to all other interested components

Figure 6.17: Management of services/components

This interface is invoked by the administrator to read/modify certain attributes, or to set the status of the service. The proxy, in turn, communicates the management request to the actual managed component. It is, however, the responsibility of the managed component to allow and process the management requests in an appropriate manner. There may be certain attributes that the component may not allow to modify; e.g., an administrator may get and set the name/location/owner of a component but attributes like serial number/product-id can only be read and cannot be modified.

Furthermore, the component has to keep its proxy object updated in the LUSs with any modifications made by an administrator. When the actual service updates its proxies in the LUSs, all the interested components get notified of the status changes. Figure 6.17 shows the typical management interactions between an admin and a managed component.

6.6 SECURITY AND SECURE LUS

One of the major deficiencies identified in the Jini architecture is insufficient security. Currently, the Jini runtime uses RMI's security mechanism to control the Jini infrastructure related activities (such as socket opening, file downloading) and it does not provide any security during service registration and lookup activities. This is a huge security hole both from the point of view of a client and a service. For example, a ***malicious*** service may easily store its *"virus"* proxy in a LUS that may get downloaded to a client with bad intentions. Similarly, a client is capable of downloading the proxy of *any service* without any authentication or accountability.

To overcome these drawbacks, a secure LUS is needed. This LUS extends from the original Jini LUS and provides all the pertinent operations (i.e., *register, lookup, notify*) but also adds certain security checks. A component passes the security interface to the Secure-LUS during its interaction and the Secure-LUS uses that interface to take necessary security measures before performing the requested operation. The *PnPSecurityInterface,* which was proposed earlier may be used for implementing a Secure-LUS.

6.7 IMPLEMENTATION

This implementation of the PnP scheme uses Jini software version 1.0 with JDK1.2. The implementation, which comprises approximately 8000 lines of Java, is designed around the notion of interfaces. Therefore, 17 Java interfaces were developed along with 28 other supporting Java classes. The classes include several utility classes that are customized for facilitating discovery, service joining, and event notification.

The implemented application models the design proposed in section 6.4 and uses Graphical User Interfaces (GUIs) to demonstrate the on-going activities of the system and to show the interaction between a client and downloaded proxy objects.

Appendix-B illustrates the Jini-based PnP implementation and presents the class hierarchy of the developed PnP application. The hierarchy shows only those classes and interfaces which are most pertinent to this application. Furthermore, the appendix shows the running PnP application along with several GUIs that correspond to the service items and downloaded service proxy objects.

6.8 CONFORMANCE TO THE REQUIREMENTS

Having designed and implemented the Jini based PnP scheme, we may now stand back to see whether this PnP application has met the major requirements for a PnP scheme that were discussed in chapter 3.

The Jini based scheme provided an easy way of *discovering* and *joining* a network. The *service advertisement* by a service provider component and *service requirement* and *service lookup* by a client component are facilitated through the help of lookup server and template matching mechanisms. A lookup server serves as a service broker and thus best fits to the requirement of a *Trader* component. Jini's distributed event model provides the *notification* mechanism that proves very useful for this application. For example, administrators get notified when services are broken, clients are notified when their corresponding modules are *upgraded*, and interested components immediately receive the information about change in any other advertised component.

The PnP application also incorporates procedures for *error detection* and *recovery*. The logging server not only provides a way to monitor system activities but also supports the *logging* requirement by saving events and messages on a disk. Jini runtime services (such as the LUS) already support the *persistence* facility through the RMI activation daemon and thus they may recover from system crashes. In a similar way, the PnP participants can be made persistent.

Security requirements are not well addressed in the current release of Jini. However, several news and mail lists indicate that the next release of Jini will incorporate better security mechanisms and secure interactions.

Finally, the combination of PnP interfaces and Jini technology interfaces provided a complete interface based PnP solution, fulfilling the interface requirements put forward in chapter 3.

6.9 EVALUATION

6.9.1 Advantages

1. Jini is an ideal technology for plug-and-play environments because it provides a *service brokering* architecture, which is most appropriate for advertising and discovering services. The architecture provides the mechanisms for service construction, lookup, communication, and use in a distributed system.
2. Jini adds an abstraction level to the networking, hiding networking details from a programmer and presenting a single, distributed memory space of objects (services).
3. The Jini infrastructure solves two difficult problems of distributed system -- discovering the network and finding the services through a directory service.
4. Software installation and upgrade are easier to perform as the driver “lives” within the device itself and the driver is distributed/supported in a plug-and-play manner.
5. Jini’s distributed leasing facility helps in *distributed garbage collection*.
6. Jini’s distributed event facility provides the important feature of *notification* that allows participant components to get notification on service arrival, service changes, and service withdrawals.
7. Jini provides a dynamic networking infrastructure, which allows systems to operate even when devices are added and removed, thus making life of a network administrator much easier.
8. Management of components is facilitated -- an administrator may manage a component/service easily through a LUS. Furthermore, a component/service may also

communicate messages/events to an interested administrator by writing certain attributes into a LUS.

9. No IDL for objects (as is the case in CORBA) is required since Jini uses the *code mobility* feature of RMI, where Java objects are “marshaled” and moved between machines. Jini, on the other hand, requires clients to use service templates for matching their services.
10. The code mobility feature for service proxy objects introduces a *de-coupling* among service provider and client components. A client does not need advanced knowledge of the service code (class) on its machine because it is the service that provides the code needed to access/use the service.
11. The service code is originally uploaded from the service provider, which keeps the client-side *synchronized* and *upgraded*.
12. Jini provides a flexible service matching mechanism, allowing clients to specify their service requirement in terms of service IDs, service interfaces, and service attributes. Moreover, the support for wild cards adds more flexibility to the matching mechanism.
13. The interface-based architecture of services benefits from the polymorphism property, allowing new services/devices to present their services with some common APIs while having their own peculiar implementation.
14. Jini benefits from the *“Write Once Run Anywhere”* nature of Java and thus works transparently in heterogeneous systems.

15. The Java-centric nature makes Jini a potential technology for Internet applications.
16. The RMI activation daemon fulfils the needs of persistence. The daemon also behaves like CORBA's Object Adapter as it enables object activation and deactivation.
17. Jini is promoted under an open source code and one can customize the code for fulfilling its needs and requirements. Furthermore, the technology may evolve fast because a useful extension/code can also be shared with the other members of Jini community.

6.9.2 Disadvantages

1. The Java-centric nature of Jini system also raises interoperability issues with non-Java application. However, efforts like RMI over IIOP may allow connecting Jini-based devices/services with other legacy applications and objects.
2. The dependence on TCP and multicast IP service also requires Jini to address issues like interoperability and bridging with non-IP networks.
3. Scalability is an issue with this technology since the technology is targeted to a network (or workgroup) of 1000 nodes in size. Therefore, the applicability of Jini in larger networks is doubtful. This problem can be handled, to some extent, with hierarchical organization.
4. Security is addressed minimally in the current release of Jini. The current release relies mainly on the existing RMI security mechanisms for its runtime operations and leaves it up to the application developers to implement their own security mechanisms.

5. The lookup server is a potential bottleneck area because if it goes down, the federation of services also collapses. However, more hierarchical and fault-tolerant systems can be built using multiple LUSs.

6. The requirements regarding storage, memory, and processor for running the associated software may limit the applicability of Jini on small devices. However, some hardware vendors have already started building custom-made chip for making device to be Jini-enabled. For example, Dallas Semiconductor Corporation has already announced the beta release for a Jini-Enabling Chip set [6] that would make a device Jini-enabled once equipped with this chip set. The chip set consists of a ROM (for JVM code and essential Java API classes), a micro-controller chip for processing, and I/O and an Ethernet interface chip for network connection.

7. Like most of the interface-based solutions, Jini requires a *priori static knowledge* of service interfaces at the client side. However, clients can use introspection and reflection features of Java to discover the functionality of an interface for which they have no knowledge.

8. *Hierarchical* and *partial* matching mechanism for services is not directly supported by Jini service architecture. Therefore, we need to manipulate the existing service matching mechanism for implementing the desired matching structure.

Chapter 7

Conclusions

This thesis has addressed an important configuration management problem related to installation, configuration, and provisioning of a new service on a network. The thesis proposed a solution to this problem by defining an auto-installing and auto-configuring plug-and-play approach for network configuration management.

The software engineering methodology of object-oriented design and analysis was used for defining the plug-and-play approach for network components. The steps followed in this methodology were -- problem definition, requirements capture, identification of core participants and assignment of responsibilities to each participant component. Furthermore, interactions among various components were related to present a high-level PnP scenario.

This thesis carried out a study of currently available distributed computing technologies, such as RMI, CORBA, DCOM, Mobile Agents, Jini, and JavaSpaces. Of the surveyed technologies, CORBA, Mobile Agents, and Jini technologies were chosen to implement and demonstrate the proposed PnP approach. The other technologies were left out due to the following reasons:

1. The study showed that RMI -- a Javatized CORBA -- is significantly limited in many respects when compared with the actual CORBA framework. For example, RMI provides a less powerful, less scalable, and static-only solution. Inter-operability,

which is a major promise of CORBA, is also not possible with RMI-based solution.

2. DCOM is the major competing technology of CORBA. Both technologies work as middleware and provide multi-language support, static/dynamic invocations, and IDL features. The major problem with DCOM is posed due to its "closed" nature. The selection of CORBA thus eliminated DCOM, as well as RMI, technology from the list of possible enabling technologies. It is, however, expected that the CORBA-based PnP approach would apply equally well in a DCOM environment as both the technological solutions are derived from the open architectures (i.e., ODP and DCE).
3. The study of JavaSpaces distributed technology revealed that it is a technology that allows sharing and co-ordination of objects using an entry-based mechanism. Since JavaSpaces itself uses Jini and is found to be more appropriate for developing applications that require a distributed persistence space for objects, the selection of Jini ruled out JavaSpaces as a separate possible candidate.

7.1 ANALYSIS OF APPLIED TECHNOLOGIES

From the design of the CORBA-based PnP approach, it was found that the CORBA framework provides a *comprehensive* solution for plug-and-play configuration management. CORBA, a result of significant standardization effort, proved to be the best choice for supporting heterogeneous environments due to its multi-platform support, application portability, and interoperability. Other important plus points of CORBA were its support for dynamic invocations and specification of several useful and common object services (such as Naming, Trading, Security, Event, Externalization, Concurrency, Transaction, etc.) that are required for distributed applications. Although, CORBA was found to be a powerful and a good technology for developing distributed applications in heterogeneous environments, its complexity and resource intensive nature limits its applicability in networks. Further, the CORBA vendors currently provide only some major and essential services (like Naming, Life Cycle, Event) in their implemented ORBs, but not those which may be essential for developing reliable network applications.

The design and implementation of the PnP approach using mobile agent technology showed that the technology provides a *flexible* and easy-to-use solution for developing network management applications. The flexibility of this approach was possible due to the push nature of the technology and easy redesign and extensibility features of mobile agent-based architectures. The technology proved very useful in scenarios related to asynchronous communication, remote information retrieval, one-to-many interactions, and navigability. Furthermore, the technology was found adaptable for supporting dynamic configuration. The limitations of this technology, however, are the lack of standardization, inter-operability, and security. These limitations are impeding the growth of this technology and that is why the technology is still in development.

Jini was found to be an ideal technology for PnP environments since it provided a *service brokering* architecture for service construction, advertisement, discovery, lookup and communication in distributed systems. Furthermore, Jini offered distributed facilities for leasing, transaction and events that enabled easy development of distributed applications. The event facility of Jini proved an important feature since it allowed the PnP participants to get notifications on service arrival, change or withdrawal, thus enabling dynamic network changes. Jini provides flexibility because it also uses the mobility of code feature. Finally, the "open source" licensing approach used by Sun Microsystems for marketing Jini is making the customization and the evolution of this technology faster than other technologies. The main drawbacks found in the Jini technology were the lack of inter-operability (because of the Java-centric nature and IP-only solution), scalability and security. Since this technology is very new[1] and still evolving, it may be too early to comment on these drawbacks.

During the analysis, it was observed that all the applied technologies bear the overhead of running supporting software on the network components. The size of software, however, varied from approach to approach. Security and Scalability are other issues that were found to be common problems. CORBA proved to be only technology, which is mature, standardized, and provides inter-operability and portability features. Although CORBA seems to provide the best solution (ideally speaking) to PnP configuration problem due to its numerous advantages, it is not chosen as the "best" candidate because "actual" ORBs are yet to attain the level that CORBA framework specifies. Furthermore, CORBA lacks important feature of code mobility that is supported by both Mobile Agent and Jini technologies.

Of all the applied technologies, Jini provided the best solution for "implementing" the proposed PnP approach. The applicability of Jini in networks, however, may be significantly limited due to its inter-operability and security problems. Therefore, it is suggested that these issues be standardized through a standard body such as the OMG.

---

[1] Jini has been in the public domain for last 3 months only.

Once the technology standardizes, becomes mature, and addresses the shortcomings identified in this thesis, it will be really applicable to this heterogeneous world of computing and networking.

Following table summarizes the analysis of the applied technologies for PnP application.

	CORBA	MA	Jini
Inter-operate	✓	✗	✗
Security	✓	Minimal	Minimal
Scalability	?	✓	✗
Portability	✓	✗	✗
Standardization	✓	✗	✗
Maturity	✓	✗	✗
Code mobility	✗	✓	✓
Resource consumption	ORB	AEE	Jini core
Interface-based	✓	✗	✓

Comparison of applied technologies

7.2 RECOMMENDATIONS FOR RESEARCHERS

This thesis provides a number of recommendations to researcher community:

1. This thesis has demonstrated the applicability of distributed technologies only in one area of network management, but researchers can extend the ideas to other areas as well. For example, the collaborative nature of mobile agents can be used for performing alarm correlation in the area of fault management.

2. The study of distributed technologies for network management purposes revealed that network management protocols have not kept up with the distributed evolution of networks. Therefore, researchers may look into evolving network management protocols and architectures to meet the requirements of distributed networks.

7.3 RECOMMENDATIONS FOR INDUSTRY

This thesis also provides some important lessons and recommendations to developers and vendors of distributed technologies:

1. The research found that none of the distributed technologies studied for this research yet provides a *complete solution* for *single area* and market. For example, Jini has been marketed for supporting consumer electronics, computer peripherals, enterprise computing, home environment, electronic commerce etc. but each of these areas may need different features such as interoperability, scalability, and ease of deployment which are not fully developed to satisfy complete requirements. This finding suggests that a better approach for the developers of these technologies would be to first target a niche market of some specific application such as enterprise networks. After providing a complete solution for a market, the reputation developed in that area can be used to facilitate introduction in the next market such as that of computer peripheral or e-commerce.

2. The research experienced “non-interoperability” as the biggest drawback of almost all

the distributed technologies[2]. Developers should resolve this issue as soon as possible by mobilizing standard bodies and consortia to evolve interoperability protocols. For instance, the JRMP protocol of RMI should be transformed to CORBA's IIOP to make it a true "Javatized CORBA".

3. Since there are a number of competing distributed technologies available in the market but none has been dominant, this finding suggests that technology developers should not only resolve the issues of interoperability but also should "leverage" each other's technologies. For instance, CORBA can be leveraged to incorporate Object-by-Value feature by using mobile agent technology.

7.4 FUTURE WORK

This research work has offered a number of research opportunities:

1. Since the thesis is based on a functional level evaluation of distributed technologies for configuration management, the work can be extended to carry out performance evaluation of these technologies in this area.
2. Security issues have been addressed and designed in this research but have not been implemented due to time constraints. Therefore, some work can be done in this direction to determine the applicability of distributed technologies for configuration management in real world applications.
3. The design provided for CORBA-based PnP approach in this research can be used as

---

[2] except CORBA

a guideline for implementing the approach for any of the available ORBs such as Orbix and VisiBroker.

4. This research has used proprietary mobile agent system (i.e., Mobile Code Toolkit), which lacked some important features such as directory service and event notification. Therefore, some other mobile agent system (e.g., Grasshopper) can be used to redesign and implement the proposed approach.
5. Although this thesis has considered a service as an “atomic entity” which can be directly used by a client, some services may be composed of many sub-services that are to be arranged and integrated dynamically during service provisioning. Therefore, this research work can be extended from “atomic service provisioning” to provide services that involve “composable service components”.
6. Finally, this research can be extended from the simple example of a network printer to other network components that are more complex in nature, such as switches and routers, to evaluate the applicability of this approach in telecommunication service management.

REFERENCES

[1] Ao, Gang; *"Software Hot Swapping Techniques"*; Technical Report SCE-98-11, Systems and Computer Engineering, Carleton University, 1998. Available at: http://www.sce.carleton.ca/netmanage/publications.html.

[2] Bieszczad, A., Raza, S. K., Pagurek, B., and White, T.; *"Agent-based Schemes for Plug-And-Play Network Components"*; in Proceedings of an International Workshop on Agents in Telecommunications Applications (IATA'98); July 1998; pages 89-101, Paris, France; available at: http://www.sce.carleton.ca/netmanage/publications.html.

[3] Carzaniga, A., Picco, G. and Vigna, G; *"Designing Distributed Applications with Mobile Code Paradigms"*; in Proceedings of the 19th International Conference on Software Engineering, pages 22-32; May 1997; Boston, USA; available from web page: http://www.elet.polimi.it/~carzanig.

[4] Chess, D., Harrison, C. and Kershenbaum, A.; *"Mobile Agents: Are they a good idea?"*; IBM Research Report; RC 19887 (88465); Intelligent Agents Project at IBM T.J. Watson Research Center; available at: http://www.research.ibm.com/iagents/paps/mobile_idea.ps.

[5] Clip, P.; *"Java's Object-Oriented Communications";* in Byte Magazine issue of February 1998; available on-line at http://www.byte.com/art/9802/sec4/art3.htm.

[6] Dallas Semiconductor Corporation; *"Dallas Semiconductor Announces First Jini-Enabling Chip Set"*; press release available at: http://www.dalsemi.com/News_Center/Press_Releases/1999/prjini.html.

[7] El-Darieby, M., Rolia, J.A.; *"Programmable Networks: A survey of current research efforts"*; available at: http://www.sce.carleton.ca/~Mohamed.El-Darieby/publications.html.

[8] Farmer, W., Guttman, J. and Swarup, V.; *"Security for mobile agents: Issues and Requirements"*; in Proceedings of the 19th National Information Systems Security Conference, pages 591-597; October 1996; Baltimore, USA; available from web page: http://www.cnri.reston.va.us/home/koe/bib/mobile.bib.html .

[9] Fuggetta, A., Picco, G. and Vigna, G.; *"Understanding Code Mobility"*; in IEEE Transactions on Software Engineering, vol. 24, no. 5, pages 342-361, 1998; available from web page: http://www.polito.it/~picco/.

[10] General Magic Inc. ; *"Mobile Agents White Paper"*; web page at: http://www.genmagic.com/technology/techwhitepaper.html.

[11] General Magic Inc.; *"Odyssey Information"*; web page at: http://www.genmagic.com/technology/odyssey.html.

[12] Goldszmidt, G. and Yemini, Y.; *"Distributed Management by Delegating Mobile Agents"*; in Proceedings of 15th International Conference on Distributed Computing Systems, June 1995, Vancouver, Canada; available at: http://www.cs.columbia.edu/~german/mbd.html.

[13] Halfhill, T.R.; *"Transforming the PC: Plug and Play";* in Byte Magazine issue of September 1994; available on-line at http://www.byte.com/art/9409/sec7/art1.htm.

[14] Hewlett-Packard Company; *"ChaiAppliance Plug and Play"*; web page at: http://www.hp.com/emso/.

[15] IBM Corporation; *"IBM Aglets Software Development Kit"*; web page at: http://www.trl.ibm.co.jp/aglets/.

[16] IKV++; *"Grasshopper: The Agent platform"*; web page at: http://www.grasshopper.de/

[17] Internet Engineering Task Force; *"Bootstrap loading using TFTP"*; RFC 906; June 1984; available at: ftp://ftp.isi.edu/in-notes/rfc906.txt.

[18] Internet Engineering Task Force; *"Dynamic Host Configuration Protocol"*; Draft Standard RFC 2131; March 1997; available at: ftp://ftp.isi.edu/in-notes/rfc2131.txt.

[19] Internet Engineering Task Force; "*Interaction between DHCP and DNS"*; Internet-Draft 'work in progress'; March 1998; available at: http://www.ietf.org/internet-drafts/draft-ietf-dhc-dhcp-dns-08.txt.

[20] Joyner, I.; *"Open Distributed Processing: Unplugged!"*; available at: http://homepages.tig.com.au/~ijoyner/ODPUnplugged.html.

[21] Lucent Technologies; *"Introducing InfernoSpaces"*; web page at: http://www.lucent-inferno.com/Pages/Developers/InfernoSpaces/ISpaces1.0/index.html.

[22] Mazumdar, S. ; *"Mapping of Common Management Information Service (CMIS) to CORBA Object Services"*; Bell Labs.; September 1996; available at: http://www.bell-labs.com/user/mazum/recent_work.html.

[23] Mazumdar, S., Swanson, K; *"WEB Based Management - CORBA/SNMP Gateway Approach";* presented at the Seventh IFIP/IEEE International Workshop on Distributed Systems: Operations & Management, October 1996, L'Aquila, Italy; available at: http://www.bell-labs.com/user/mazum/recent_work.html.

[24] Microsoft Corporation; *"DCOM Architecture"*; white paper available at: http://www.microsoft.com/com/comPapers.asp#DCOMPapers.

[25] Microsoft Corporation; "*Home Networking – Universal Plug and Play Primer"*; web page at: http://www.microsoft.com/homenet/upnp.htm.

[26] Microsoft Corporation; "*Microsoft Windows and the Plug and Play Framework Architecture"*; March 1994; web page at http://www.microsoft.com/win32dev/base/pnp.htm.

[27] Microsoft Corporation; "*The Millenium Research Project"*; web site at: http://www.research.microsoft.com/sn/Millennium/.

[28] Object Management Group; *"CORBA services: Common Object Services Specification"*; December 1998; available at: http://www.omg.org/library/csindx.html.

[29] Object Management Group; *"Mobile Agent System Interoperability Facilities Specification"*; OMG TC Document orbos/98-03-09.pdf; available at: ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf.

[30] Object Management Group; *"The Common Object Request Broker: Architecture and Specification"*; CORBA/IIOP 2.2, February 1998; available at:

http://www.omg.org/library/c2indx.html.

[31] Object Management Group; *“UML Notation Guide”;* ad/97-08-05 OMG TC Work in Progress; available at: http://www.omg.org/techprocess/meetings/schedule/Technology_Adoptions.html#tbl_UML_Specification.

[32] ObjectSpace Inc.; *“Voyager”*; web page at: http://www.objectspace.com/Products/voyager1.htm.

[33] Open Software Foundation; *“DCE Documentation Pointers”;* available at: http://www.opengroup.org/tech/dce/info/docptrs.htm.

[34] Perpetuum Mobile Procura project; *“Mobile Code Toolkit”*; Version 1.0 (1997); Network Management Laboratory, Carleton University, Ottawa; available at: http://www.sce.carleton.ca/netmanage/mctoolkit.

[35] Raza, S. K., Bieszczad, A.; *“Network Configuration with Plug and Play Components”*; to be presented at the 6th IFIP/IEEE International Symposium on Integrated Network Management (IM’99), May 1999, Boston, USA; available at: http://www.sce.carleton.ca/netmanage/publications.html.

[36] Stallings, W.; *“SNMP, SNMPv2, and CMIP: The practical guide to Network Management standards”;* Addison-Wesley publishing company, 1993.

[37] Sun Microsystems Inc.; *“Java Remote Method Invocation Specification”;* available at: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[38] Sun Microsystems Inc.; *“JavaSpaces Specification”*; available at: http://www.sun.com/jini/specs/.

[39] Sun Microsystems Inc.; *“Jini Architecture Specification”*; available at: http://www.sun.com/jini/specs/.

[40] Sun Microsystems Inc.; *“Java Language Overview”;* white paper available at: http://java.sun.com/docs/white/index.html.

[41] Sun Microsystems Inc.; *“Java RMI”;* white paper available at: http://java.sun.com/products/javaspaces/whitepapers/index.html.

[42] Sun Microsystems Inc.; *“Jini Architectural Overview”*; technical white paper available at: http://www.sun.com/jini/whitepapers/.

[43] Sun Microsystems Inc.; *“Jini Device Architecture Specification”*; available at: http://www.sun.com/jini/specs/.

[44] Sun Microsystems Inc.; *“Jini Discovery and Join Specification”*; available at: http://www.sun.com/jini/specs/.

[45] Sun Microsystems Inc.; *“Jini Distributed Event Specification”*; available at: http://www.sun.com/jini/specs/.

[46] Sun Microsystems Inc.; *“Jini Distributed Leasing Specification”*; available at: http://www.sun.com/jini/specs/.

[47] Sun Microsystems Inc.; *"Jini Lookup Service Specification"*; available at: http://www.sun.com/jini/specs/.

[48] Sun Microsystems Inc.; *"Jini Transaction Specification"*; available at: http://www.sun.com/jini/specs/.

[49] Susilo, G., Bieszczad, A. and Pagurek, B.; *"Infrastructure for Advanced Network Management based on Mobile Code"*; in Proceedings of the IEEE/IFIP Network Operations and Management Symposium NOMS'98; February 1998; New Orleans, USA; available at: http://www.sce.carleton.ca/netmanage/publications.html.

[50] Umar, A.; *"Distributed computing; a practical synthesis of networks, client-server systems, distributed applications, and open systems"*; Prentice Hall, 1993.

[51] University of Stuttgart, Distributed Systems Group; *"The Home of the MOLE"*; web page at: http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html.

[52] Wooldridge, M, and Jennings, N.R.; *"Pitfalls of Agent-Oriented Development"*; in Proceedings of 2nd International Conference on Autonomous Agents, May 1998; St. Paul, USA.

[53] Yale Linda Group; *"Linda system"*; various technical reports available at: http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html.

Appendix A

Implementation of Mobile Agent based PnP scheme

This appendix presents the class hierarchy and the running application of the mobile agents based PnP approach implementation.

Class Hierarchy

Since the mobile agents based PnP implementation uses Mobile Code Toolkit (described in section 5.2.4), the class hierarchy shows several classes that are proprietary to the employed toolkit. Readers may refer [34] for further information on these proprietary classes.

The class hierarchy mainly comprises two types of implemented classes – Virtual Managed Component (VMC) classes and Agent classes. The VMC classes reside on network components to provide controlled-access to the underlying component, whereas the agent classes represent mobile agents that are injected by a component. These agents migrate from component to component and interact with VMCs to achieve the delegated tasks. Both the VMC and agent classes implement several interfaces that are required for the functionality of the toolkit and the PnP application. All of the implemented VMCs inherit from the toolkit's generic VMC class (i.e., *VirtualManagedComponent*), while all the agents extend and customize the toolkit's generic agent class (i.e., *Netlet*). In the toolkit, both the VMC and Netlet classes inherit from a super class called the *MobileCodeObject*. The VMCs and agents that are shown in Figure a.1 have been described in detail in chapter 5.

The diagram also shows some other supporting classes that require brief explanation here. *DiscoveryListener*, a helper class for SRD-Agent of section 5.3.6.4, runs on the PnP-VMC and receives the discovery information sent by a SRD agent over a socket connection. Another interface, the *VMCProp*, was also defined in this implementation for passing parameters to a VMC through property files that also required modification in the *MobileCodeManager* class of the toolkit. In this implementation, another important class was developed, called the *MCParam*, to pass the parameters to the mobile agents during their construction and instantiation through the PnP-VMC.

Running Application

Following figure a.2 shows a screen-shot of the application run for the mobile-agent based PnP scheme. The diagram shows four separate consoles that represent four components – a PnP component, a vendor component, and two client components. In this implementation, a printer was chosen as an example of the PnP component, whereas a PC and a Sun WS (running Windows NT4.0 and Solaris 2.2 operating systems respectively) represented the clients. Since these consoles are the *agent host*, they run Mobile Code Daemon (MCD) that provides the required mobility infrastructure. In short, every component is represented as a console and the component software actually runs over a MCD. All the components, shown in the diagram, also have respective VMCs installed on them (as described in section 5.3.3) that are accessed by various visiting agents through well-defined interfaces.

The output consoles of the figure illustrate some of the steps in the PnP scenario by displaying various messages from the VMCs and visiting mobile agents.

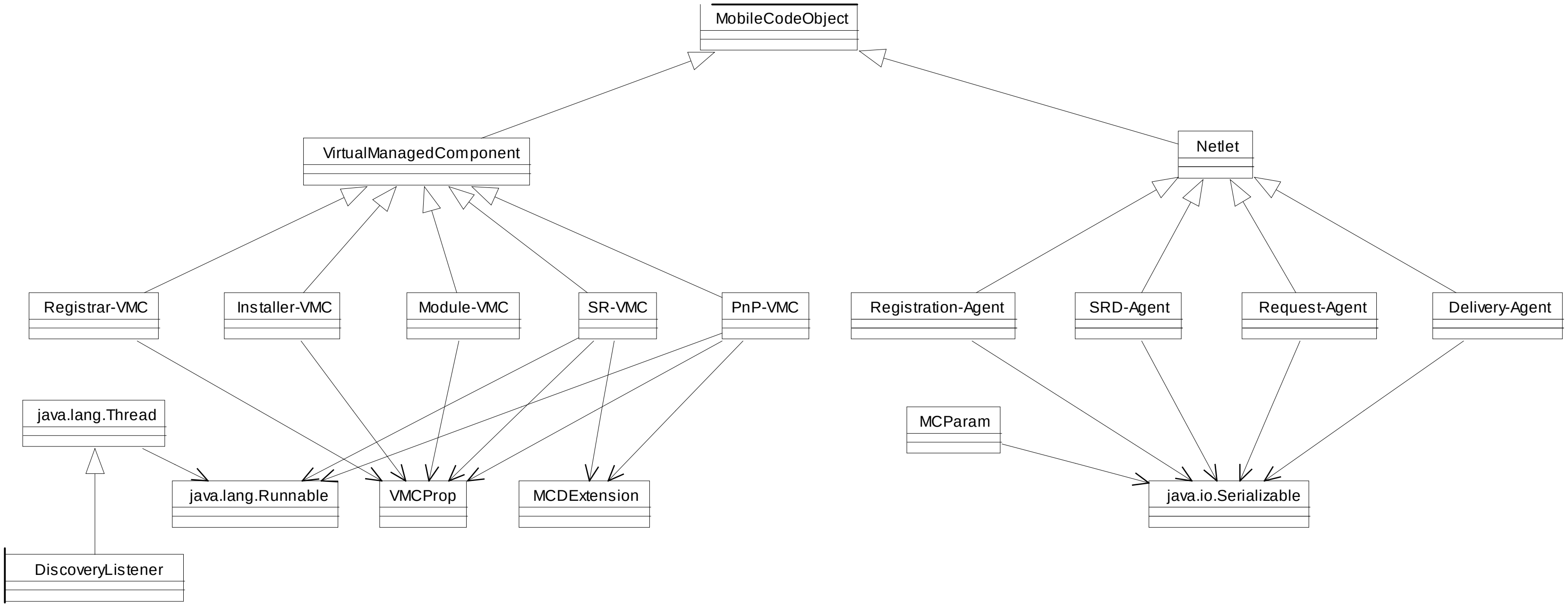
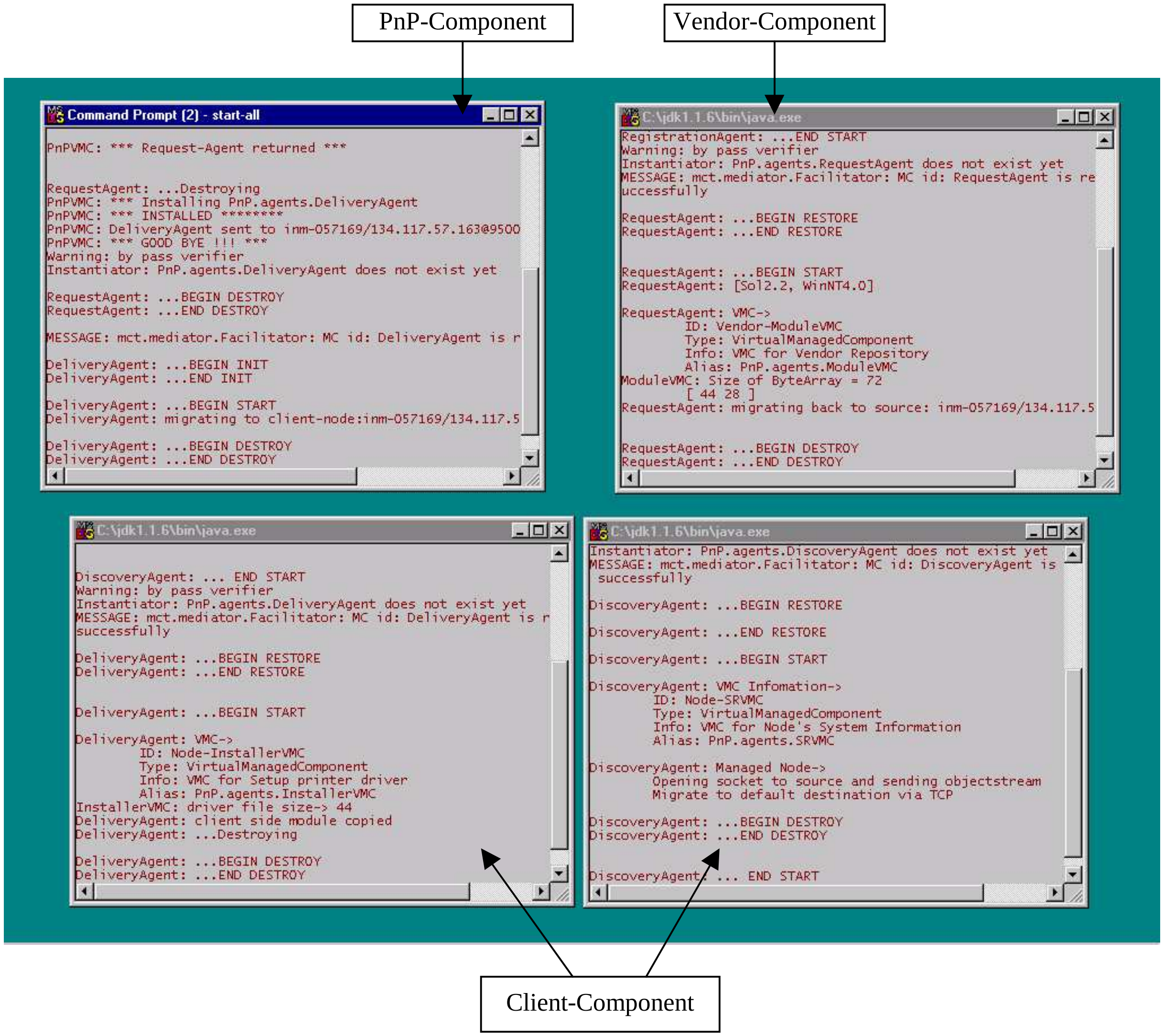


Figure a.1: Class hierarchy



```
PnPVMC: *** Request-Agent returned ***


RequestAgent: ...Destroying
PnPVMC: *** Installing PnP.agents.DeliveryAgent
PnPVMC: *** INSTALLED ********
PnPVMC: DeliveryAgent sent to inm-057169/134.117.57.163@9500
PnPVMC: *** GOOD BYE !!! ***
Warning: by pass verifier
Instantiator: PnP.agents.DeliveryAgent does not exist yet

RequestAgent: ...BEGIN DESTROY
RequestAgent: ...END DESTROY

MESSAGE: mct.mediator.Facilitator: MC id: DeliveryAgent is r

DeliveryAgent: ...BEGIN INIT
DeliveryAgent: ...END INIT

DeliveryAgent: ...BEGIN START
DeliveryAgent: migrating to client-node:inm-057169/134.117.5

DeliveryAgent: ...BEGIN DESTROY
DeliveryAgent: ...END DESTROY
```

```
RegistrationAgent: ...END START
Warning: by pass verifier
Instantiator: PnP.agents.RequestAgent does not exist yet
MESSAGE: mct.mediator.Facilitator: MC id: RequestAgent is re
uccessfully

RequestAgent: ...BEGIN RESTORE
RequestAgent: ...END RESTORE


RequestAgent: ...BEGIN START
RequestAgent: [Sol2.2, WinNT4.0]

RequestAgent: VMC->
        ID: Vendor-ModuleVMC
        Type: VirtualManagedComponent
        Info: VMC for Vendor Repository
        Alias: PnP.agents.ModuleVMC
ModuleVMC: Size of ByteArray = 72
        [ 44 28 ]
RequestAgent: migrating back to source: inm-057169/134.117.5


RequestAgent: ...BEGIN DESTROY
RequestAgent: ...END DESTROY
```

```


DiscoveryAgent: ... END START
Warning: by pass verifier
Instantiator: PnP.agents.DeliveryAgent does not exist yet
MESSAGE: mct.mediator.Facilitator: MC id: DeliveryAgent is r
successfully

DeliveryAgent: ...BEGIN RESTORE
DeliveryAgent: ...END RESTORE


DeliveryAgent: ...BEGIN START

DeliveryAgent: VMC->
        ID: Node-InstallerVMC
        Type: VirtualManagedComponent
        Info: VMC for Setup printer driver
        Alias: PnP.agents.InstallerVMC
InstallerVMC: driver file size-> 44
DeliveryAgent: client side module copied
DeliveryAgent: ...Destroying

DeliveryAgent: ...BEGIN DESTROY
DeliveryAgent: ...END DESTROY
```

```
Instantiator: PnP.agents.DiscoveryAgent does not exist yet
MESSAGE: mct.mediator.Facilitator: MC id: DiscoveryAgent is
 successfully

DiscoveryAgent: ...BEGIN RESTORE

DiscoveryAgent: ...END RESTORE

DiscoveryAgent: ...BEGIN START

DiscoveryAgent: VMC Infomation->
        ID: Node-SRVMC
        Type: VirtualManagedComponent
        Info: VMC for Node's System Information
        Alias: PnP.agents.SRVMC

DiscoveryAgent: Managed Node->
        Opening socket to source and sending objectstream
        Migrate to default destination via TCP

DiscoveryAgent: ...BEGIN DESTROY
DiscoveryAgent: ...END DESTROY


DiscoveryAgent: ... END START
```

Figure a.2: Running PnP application

Appendix B

Implementation of Jini based PnP scheme

This appendix describes the class hierarchy, application run, and various GUIs of Jini-based PnP implementation.

Class Hierarchy

The class hierarchy diagram for the implemented Jini-based scheme is shown in Figure b.1. Although several classes and interfaces were developed in Jini-based implementation, this diagram shows only the pertinent classes and interfaces. The class diagram comprises several kind of classes – classes for PnP participant components (i.e., PnP, Client, and Vendor), component interface, classes for components' proxy objects and user interface classes. To understand the relationship among these classes, let us take a component as an example.

Consider the *PnP-Printer* class that is a subclass of the *PnPComponent*, where the *PnPComponent* class itself extends from another abstract class -- *NetworkComponent*. A PnP-Printer object advertises its proxy object, called the *PnP-Proxy*, in a LUS that is downloaded by *ClientComponent* objects. This proxy implements various important interfaces such as *BasicPrinterInterface*, *ManagementInterface* and *ServiceUIInterface*, as discussed in section 6.4.4. The proxy object also "contains" two UIs (applets) – *PrinterManagement* and *PrinterInteraction* – that allow a user to interact visually. The PnP-Printer object represents the actual service component that provides the service and thus implements the appropriate remote interfaces. Furthermore, the printer object incorporates several other interfaces that are required for the event registration and notification mechanisms through the helper objects such as *VendorEventListener* and *UpgradeEventListener*. The class hierarchy can be explored in a similar manner for other participant components.

Running Application

Figure b.2 shows a screen-shot of the running application of Jini-based PnP scheme. The diagram shows many consoles representing various participant components -- an http server (class server) application, the RMI activation daemon, and application consoles associated with Admin, Client, PnP, and Vendor components. The application consoles are running a JVM within which the actual application is being executed.

The figure also shows several user interfaces (applets). These interfaces are associated with the service proxy objects that get downloaded on a client, thus allowing a user to interact visually. The examples of such UIs are – interaction applets, management applets and service module applets.

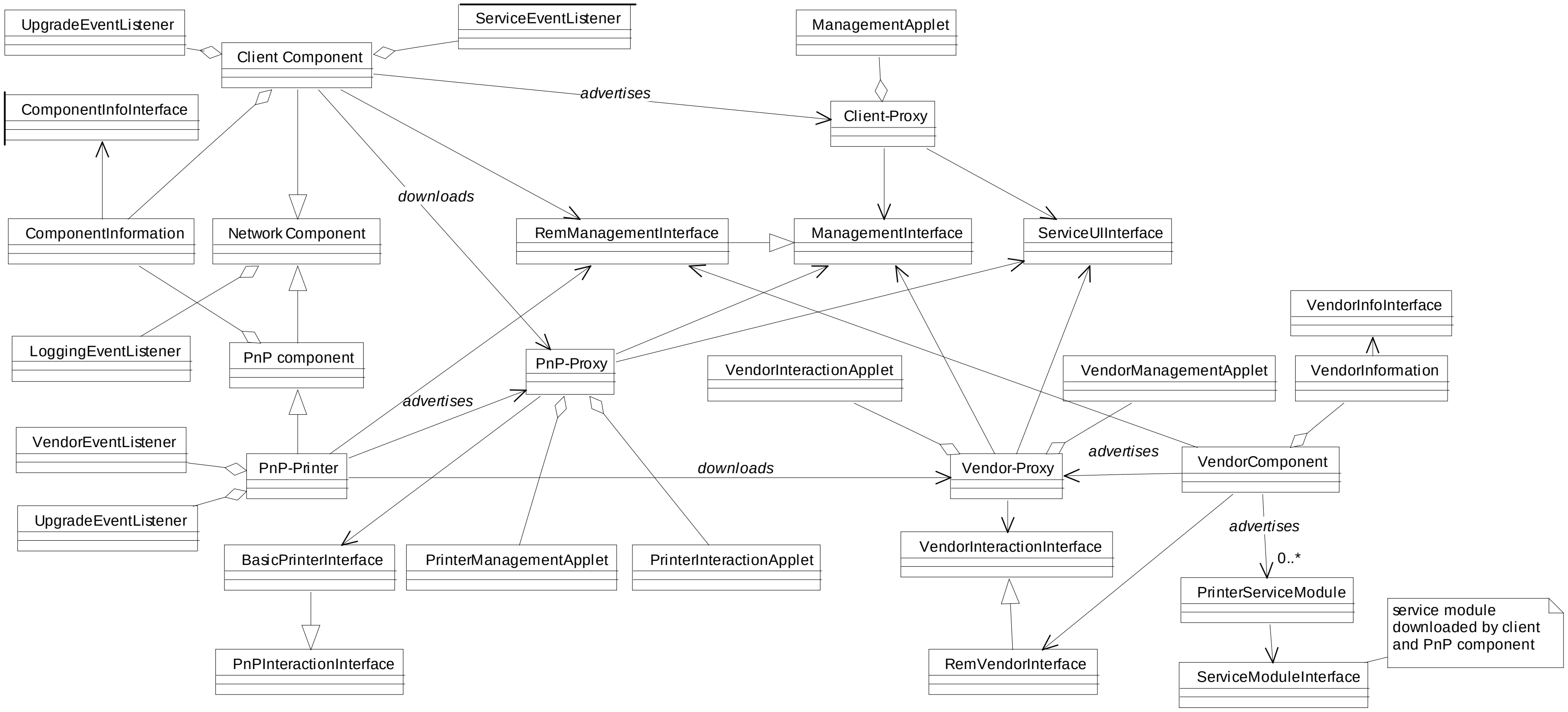
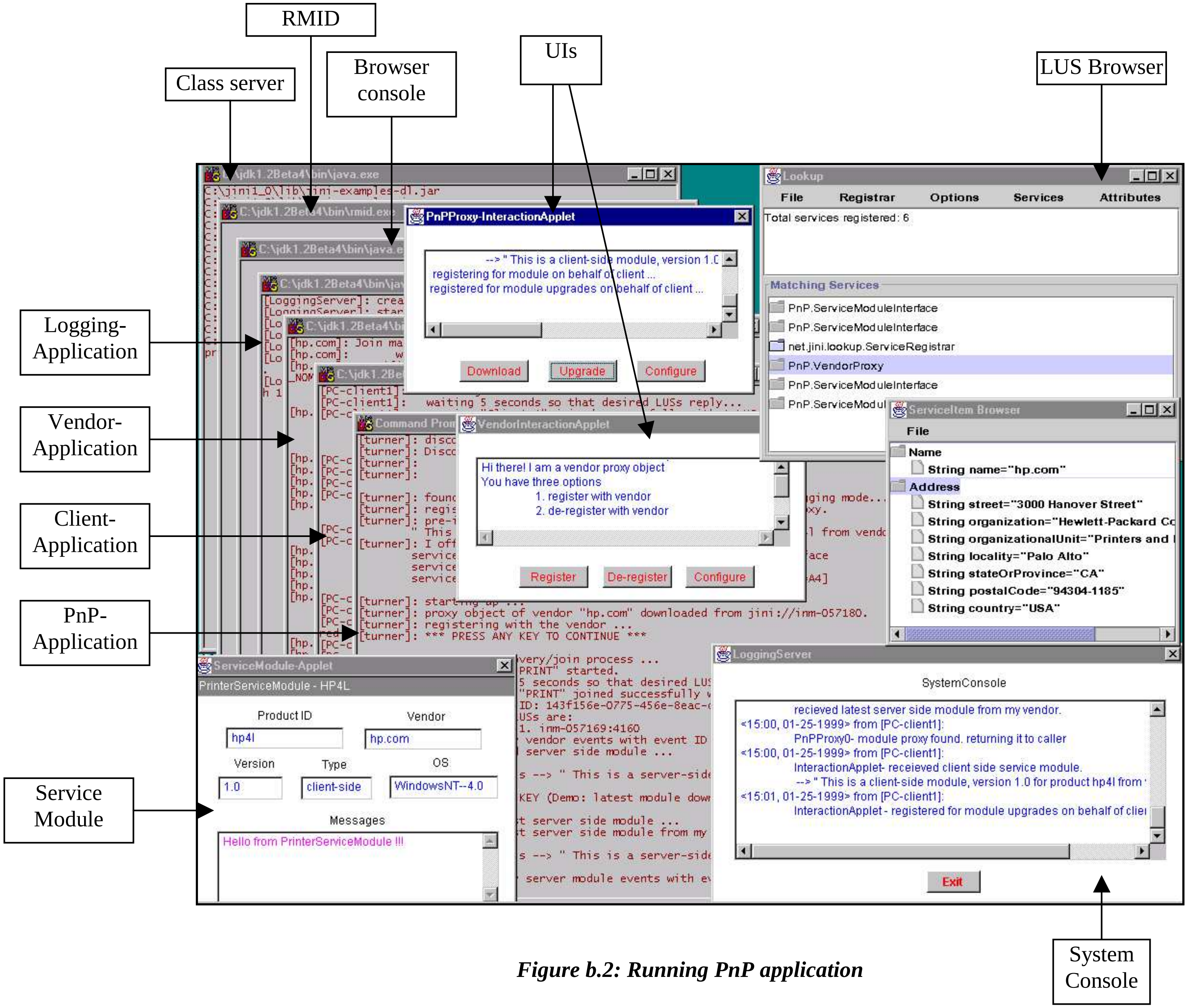


Figure b.1: Class hierarchy



```
C:\jini1_0\lib\jini-examples-dl.jar
[LoggingServer]: crea
[LoggingServer]: star
[hp.com]: Join ma
[hp.com]:        w
[PC-client1]:
[PC-client1]:   waiting 5 seconds so that desired LUSs reply...
[turner]: disco
[turner]: Disco
[turner]:
[turner]:
[turner]: founc
[turner]: regis
[turner]: pre-i
        " This
[turner]: I off
        service
        service
        service
[turner]: starting up ...
[turner]: proxy object of vendor "hp.com" downloaded from jini://inm-057180.
[turner]: registering with the vendor ...
[turner]: *** PRESS ANY KEY TO CONTINUE ***
```

```
ging mode...
xy.
l from vendo
ace
A4]
```

```
very/join process ...
PRINT" started.
5 seconds so that desired LUS
"PRINT" joined successfully w
ID: 143f156e-0775-456e-8eac-
USs are:
1. inm-057169:4160
 vendor events with event ID
 server side module ...

s --> " This is a server-side

KEY (Demo: latest module down

t server side module ...
t server side module from my

s --> " This is a server-side

 server module events with ev
```

--> " This is a client-side module, version 1.0
registering for module on behalf of client ...
registered for module upgrades on behalf of client ...

Hi there! I am a vendor proxy object
You have three options
1. register with vendor
2. de-register with vendor

Total services registered: 6

Matching Services

- PnP.ServiceModuleInterface
- PnP.ServiceModuleInterface
- net.jini.lookup.ServiceRegistrar
- PnP.VendorProxy
- PnP.ServiceModuleInterface
- PnP.ServiceModul

- Name
  - String name="hp.com"
- Address
  - String street="3000 Hanover Street"
  - String organization="Hewlett-Packard Co
  - String organizationalUnit="Printers and
  - String locality="Palo Alto"
  - String stateOrProvince="CA"
  - String postalCode="94304-1185"
  - String country="USA"

PrinterServiceModule - HP4L

Product ID: hp4l

Vendor: hp.com

Version: 1.0

Type: client-side

OS: WindowsNT--4.0

Messages: Hello from PrinterServiceModule !!!

```
        recieved latest server side module from my vendor.
<15:00, 01-25-1999> from [PC-client1]:
        PnPProxy0- module proxy found. returning it to caller
<15:00, 01-25-1999> from [PC-client1]:
        InteractionApplet- receieved client side service module.
         --> " This is a client-side module, version 1.0 for product hp4l from
<15:01, 01-25-1999> from [PC-client1]:
        InteractionApplet - registered for module upgrades on behalf of clie
```

Figure b.2: Running PnP application

User Interfaces

The Jini-based implemented PnP application involves various user interfaces. There are two types of user interfaces – the interfaces that are supplied along with the Jini software, and the interfaces that are developed for this PnP application.

The following pages display and explain the contents of each LUS, in addition to the description of other implemented GUIs.

LUS browser

Jini software provides a tool, called the '*Service Browser GUI*', that performs a discovery of all the available LUS (within the multicast range) on a network and shows the discovered LUSs. A user may select a LUS to traverse and browse its contents for the registered services using this GUI. The browser GUI shown in figure b.3 illustrates that there are four nearby LUSs (i.e., *Registrar* in the figure) running on four different machines (i.e., inm-057178, pmpserver, etc.). It is to be noted that the selected LUS (i.e.,

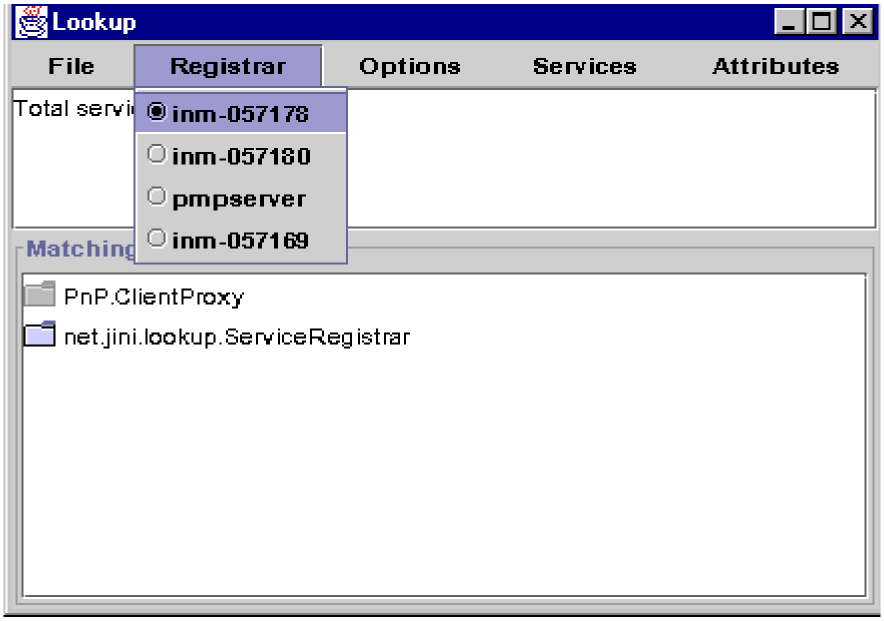
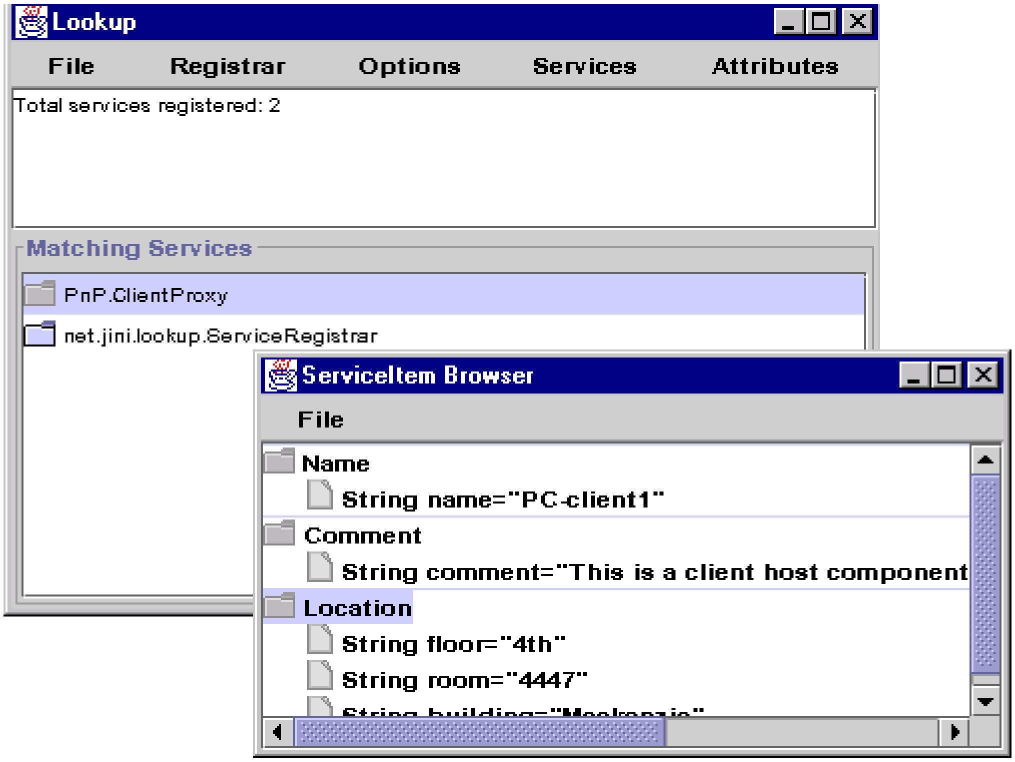


Figure b.3: Four LUSs running on four different machines

inm-057178) is currently holding two service proxy objects – a proxy for the LUS itself (i.e., *ServiceRegistrar*) and a proxy for a client component (i.e., *ClientProxy*). A user may also browse associated service attributes of a registered service by using other menus of the browser, such as attributes and services menus.

Clients-LUS

Figure b.4 shows a clients-LUS and an advertised client item. The item comprises – a *ClientProxy* and attributes. The figure also shows client item's attributes (i.e., Name, Comment, and Location) with the help of 'ServiceItem Browser' facility.



Total services registered: 2

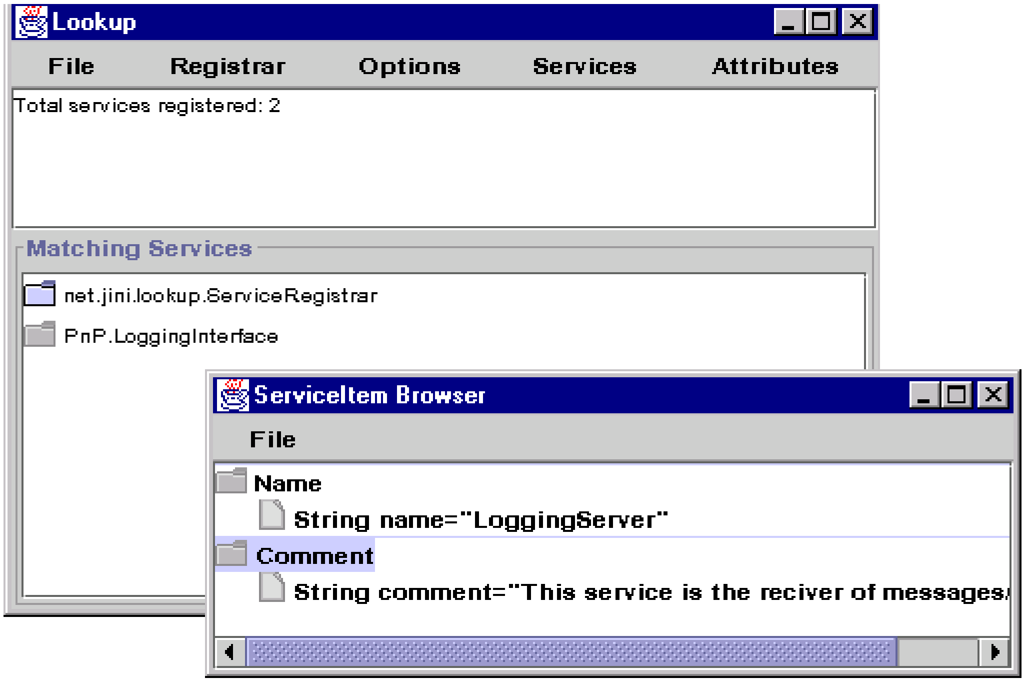
PnP.ClientProxy
net.jini.lookup.ServiceRegistrar

Name
String name="PC-client1"
Comment
String comment="This is a client host component
Location
String floor="4th"
String room="4447"

Figure b.4: Clients-LUS and advertised Client-Item

Admin-LUS

A logging item is advertised in an admin-LUS by a logging service. The item comprises a *LoggingProxy* (which implements *LoggingInterface*) and attributes (Name and Comment) as shown in figure b.5.



Total services registered: 2

net.jini.lookup.ServiceRegistrar

PnP.LoggingInterface

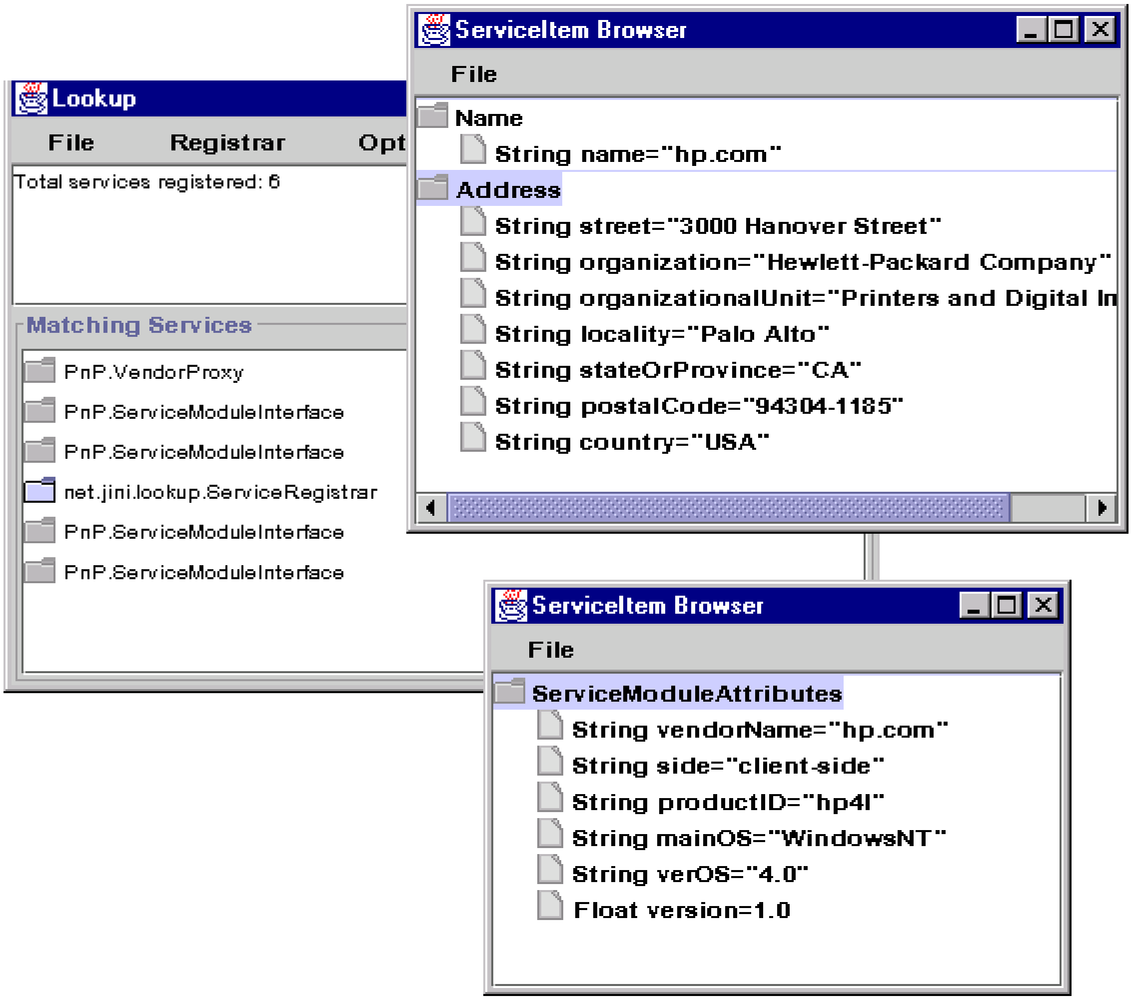
String name="LoggingServer"

String comment="This service is the reciver of messages

Figure b.5: Admin-LUS and advertised Logging-Item

Vendor-LUS

Section 6.5.2 has described the vendor setup in which a vendor advertises its own proxy object and several proxy objects for various service modules in the vendor-LUS. Following figure b.6 shows the contents of the vendor LUS that comprises a *VendorProxy* object and four *ServiceModule* objects (with *ServiceModuleInterface* interface). The modules that are shown in the figure are of two types – server and client -- and correspond to various supported products.



Total services registered: 6

Matching Services

- PnP.VendorProxy
- PnP.ServiceModuleInterface
- PnP.ServiceModuleInterface
- net.jini.lookup.ServiceRegistrar
- PnP.ServiceModuleInterface
- PnP.ServiceModuleInterface

```
Name
    String name="hp.com"
Address
    String street="3000 Hanover Street"
    String organization="Hewlett-Packard Company"
    String organizationalUnit="Printers and Digital In
    String locality="Palo Alto"
    String stateOrProvince="CA"
    String postalCode="94304-1185"
    String country="USA"
```

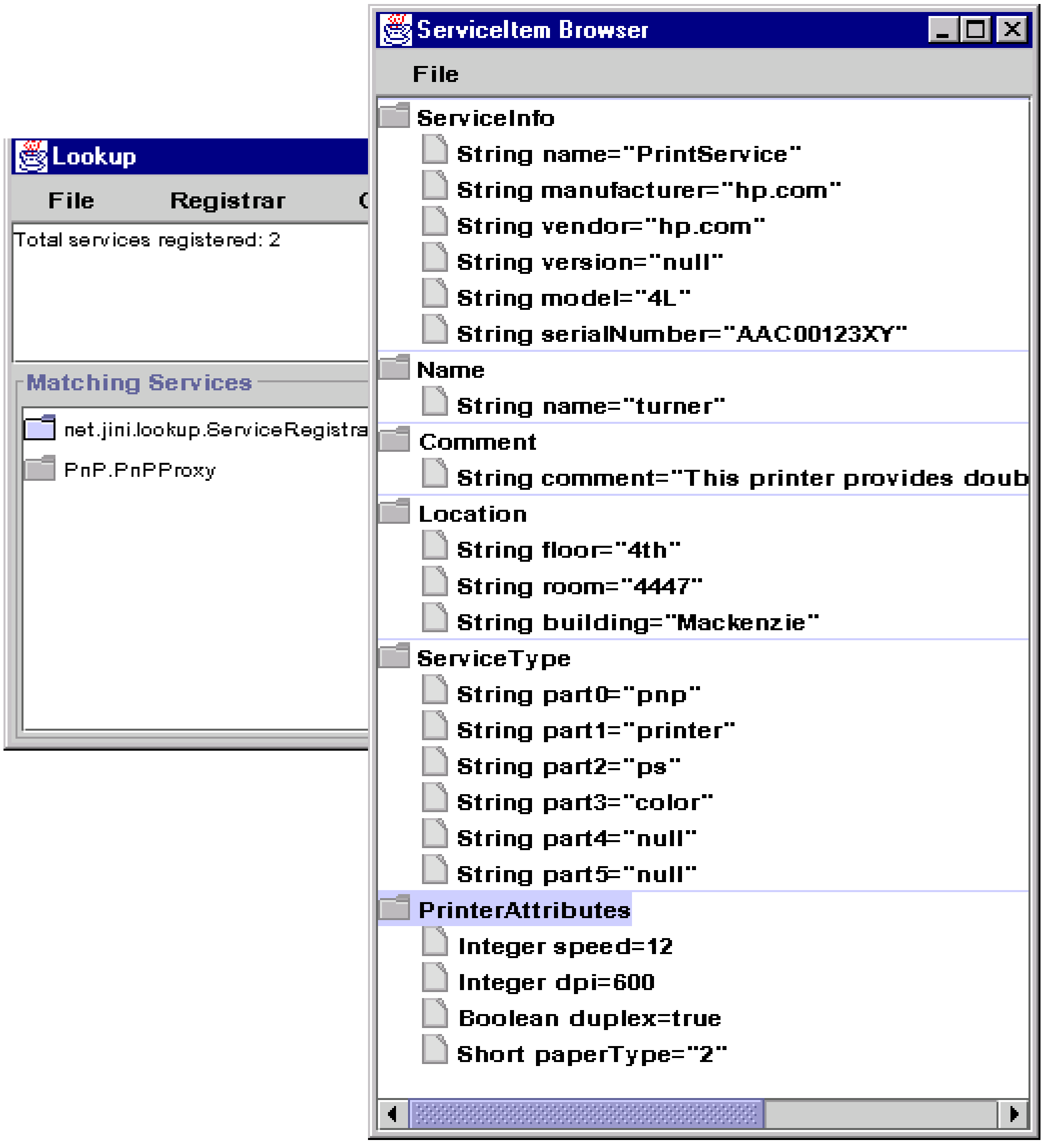
```
ServiceModuleAttributes
    String vendorName="hp.com"
    String side="client-side"
    String productID="hp4l"
    String mainOS="WindowsNT"
    String verOS="4.0"
    Float version=1.0
```

Figure b.6: Vendor-LUS and advertised Vendor-Item and Module-Items

The figure also shows the attributes associated with each advertised object. For example, a name and address attributes are stored with the vendor proxy object, whereas attributes like vendor name, OS type, version, etc. are associated with an advertised module. The module attributes are more important because they are used to match a client's request to a specific module type.

<u>Printers-LUS</u>

Figure b.7 presents a view of the printers-LUS that stores advertised printer proxy objects. The figure shows that currently there is only one advertised printer service, represented by the *PnP-Proxy* object. As described in sections 6.5.4 and 6.5.5, a PnP component advertises its proxy with several attributes and clients use these attributes to find their matching services. The pertinent attributes include – service information, name, location, comment, type, and device-specific (printer) attributes. Note that the service type attribute, along with the Printer specific attributes, helps clients in matching services



Total services registered: 2

```
net.jini.lookup.ServiceRegistra
PnP.PnPProxy
```

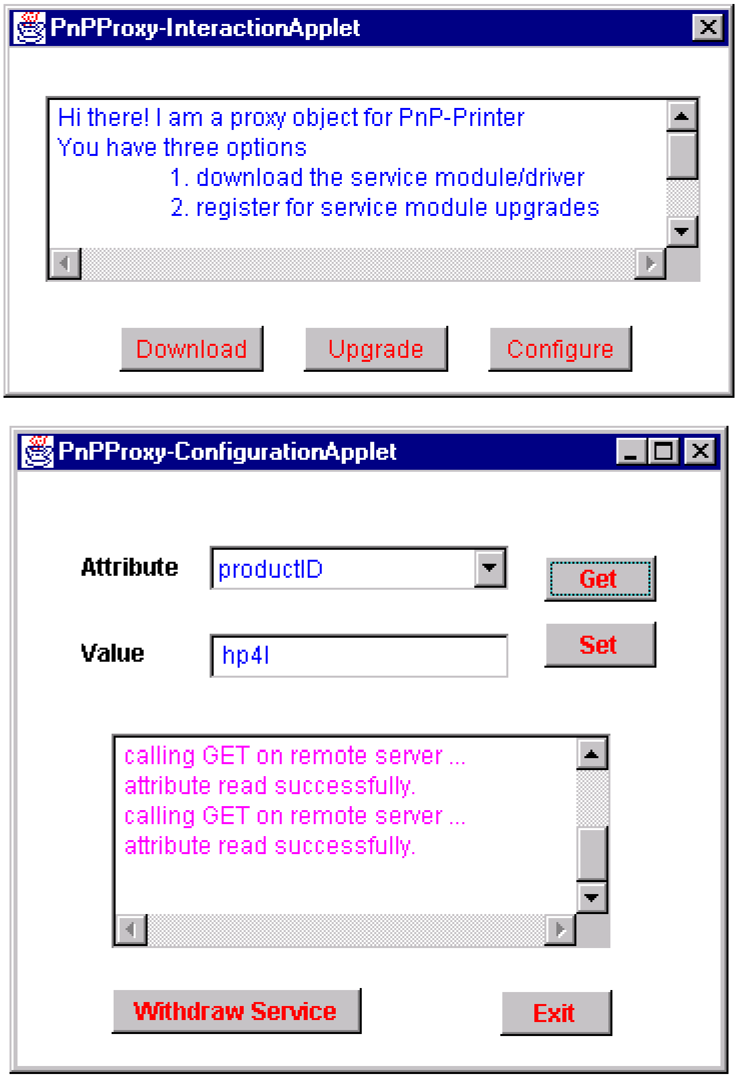
```
ServiceInfo
    String name="PrintService"
    String manufacturer="hp.com"
    String vendor="hp.com"
    String version="null"
    String model="4L"
    String serialNumber="AAC00123XY"
Name
    String name="turner"
Comment
    String comment="This printer provides doub
Location
    String floor="4th"
    String room="4447"
    String building="Mackenzie"
ServiceType
    String part0="pnp"
    String part1="printer"
    String part2="ps"
    String part3="color"
    String part4="null"
    String part5="null"
PrinterAttributes
    Integer speed=12
    Integer dpi=600
    Boolean duplex=true
    Short paperType="2"
```

Figure b.7: Printers-LUS and advertised PnP-Item

using a hierarchical and sub-typing matching mechanism.

PnP component's UI elements

As discussed in section 6.4.2, every advertised proxy object implements the *ServiceUIInterface*, which allows user to interact visually. Following figure shows two applets that actually provide a visual interface to a user and an administrator for interaction and management purposes respectively. A PnP-Proxy object, which is



Hi there! I am a proxy object for PnP-Printer
You have three options
1. download the service module/driver
2. register for service module upgrades

Attribute productID

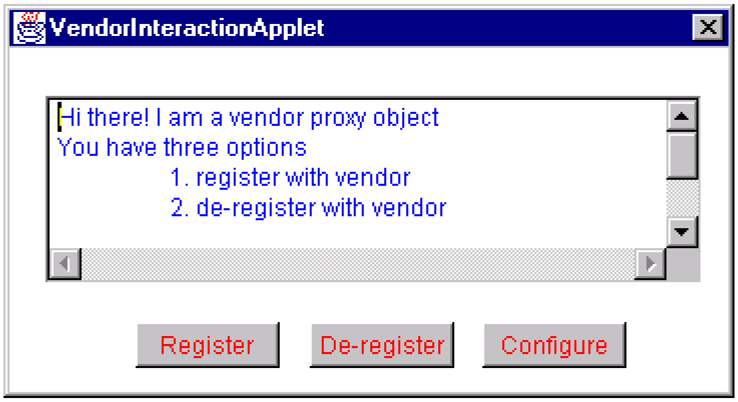
Value hp4l

calling GET on remote server ...
attribute read successfully.
calling GET on remote server ...
attribute read successfully.

Figure b.8: PnP proxy's Interaction and Management Applets downloaded to a client machine from a LUS, provides these interfaces (applets).

Vendor's UI elements

Similar to a PnP-Proxy object, a vendor-proxy object also provides an Interaction applet which allows interaction among a PnP component and a vendor. Although not implemented in this application, the vendor proxy object may also supply a management applet.

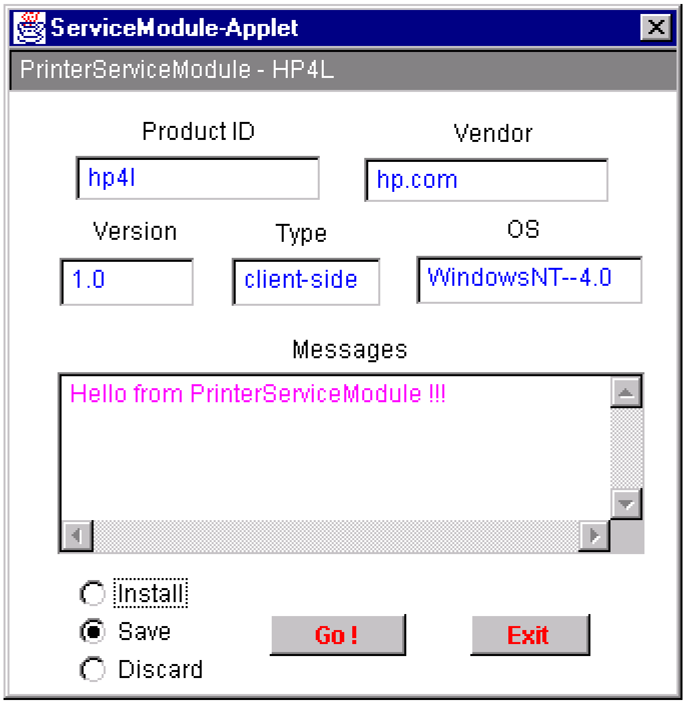


Hi there! I am a vendor proxy object
You have three options
1. register with vendor
2. de-register with vendor

Figure b.9: Interaction Applet of downloaded Vendor-Proxy

Service Module

A service module is also represented as an applet in the Jini-based PnP implementation. These modules get downloaded both on the client and PnP components. These components, in turn, use the module's GUI to install the actual encapsulated service module.



PrinterServiceModule - HP4L

Product ID: hp4l
Vendor: hp.com
Version: 1.0
Type: client-side
OS: WindowsNT--4.0

Messages

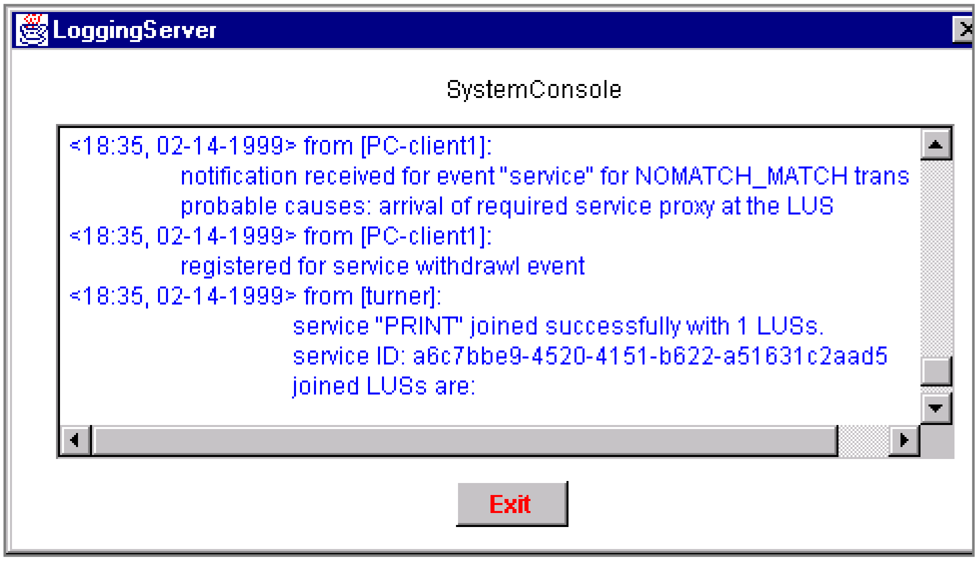
```
Hello from PrinterServiceModule !!!
```

○ Install
● Save
○ Discard

Figure b.10: UI for downloaded service module

System Console

A GUI is also implemented to display system-related activities and error messages from various PnP participants. This system console is updated by the logging service and it helps an administrator in keeping track of the on-going activities in the system.



```
<18:35, 02-14-1999> from [PC-client1]:
          notification received for event "service" for NOMATCH_MATCH trans
          probable causes: arrival of required service proxy at the LUS
<18:35, 02-14-1999> from [PC-client1]:
          registered for service withdrawl event
<18:35, 02-14-1999> from [turner]:
                    service "PRINT" joined successfully with 1 LUSs.
                    service ID: a6c7bbe9-4520-4151-b622-a51631c2aad5
                    joined LUSs are:
```

Figure b.11: Logging application