An Overview of Jini Technology and Its Affect on Distributed Computing

Johannes Lehto

11th of December 2000

University of Joensuu

Computer Science

Master's Thesis

Preface

This thesis was initiated at Network Terminals Department of Nokia Networks and finished at Nokia Entrepreneurial Web Department of Nokia Ventures Organization. I am pleased to have been able to work with this subject I am personally interested in.

I thank my original mentor Jari Torkkel for encouraging me to start studying this subject, and Raimo Bäckström for motivating me to finish my thesis. I also thank Jussi Parkkinen from the Department of Computer Science, University of Joensuu, for accepting this subject and providing comments on scientific writing and structure of the thesis. Special thanks to Marja-Liisa Lehto, who helped me a lot with my English.

Johannes Lehto
Helsinki, 11th of December 2000

Abstract

This thesis focuses on introducing the Jini technology on the general level and presenting an understanding of its programming models. It will also provide some ideas of how this new technology could be used in the future and for what purposes. Moreover, some problems related to the designing of distributed systems are discussed, as well as the solutions to them offered by the Jini technology.

Although Jini is build atop the Java programming language, anybody wanting to use Jini to build distributed systems has to change his or her way of thinking quite radically. Traditional distributed system development tries to hide the network layer from both the application and the developer. Jini model does not do that, because it requires that the developer really thinks about issues like partial network failures, and it provides a framework to build and design reliable and robust network services.

Right now the industry is moving from the disc centric approach to the network centric computing. The network will be the central connecting tissue and the notion of applications and peripherals will be replaced by network-available services and their consumers.

One of the greatest changes in the development of distributed systems is that traditionally data has come to the code, which uses it. With Jini, it is possible to move objects over the network. It means that both the code and the data are transferred. This model allows flexible building of new kinds of services in the network.

In addition, one of the purposes of this thesis is to present what the minimum requirements for Jini-enabled devices are and what Java features must be supported to make a minimum Jini system work within low-capacity embedded devices. Moreover, some drawbacks of this technology are discussed. Some assumptions about the future of ubiquitous computing are also made.

Table of Contents

PREFACE

ABSTRACT

TABLE OF CONTENTS

ABBREVIATIONS

1. INTRODUCTION 1

2. AN OVERVIEW OF DISTRIBUTED SYSTEMS 4

2.1 Anatomy and Features of Distributed Systems 4

2.2 Major Problems and the Model of Unified Objects 6

2.3 Demands of Distributed Systems 10

2.4 Java's Affect on Difficulties 15

3. WHAT IS JINI? 21

3.1 The Basic Idea 21

3.2 Goals 22

3.3 Java Platform 23

3.4 Lookup 24

3.5 Proxy 25

3.6 Discovering Services 26

3.7 System Requirements 26

4. BASIC CONCEPTS OF JINI 28

4.1 Discovering and Joining Jini Communities 28

4.1.1 Discovering Lookup Services 29

4.1.2 Announcing the Existence of the Lookup Services 33

4.1.3 Unicast Discovery Protocol 35

4.1.4 Join Protocol 37

4.2 Lookup Service 40

4.2.1 The Use of Lookup Service 40

4.2.2 Design Issues 43

4.3 Leasing 47

4.3.1 Benefits of Leasing 48

4.3.2 Design Issues 49
4.3.3 Managing a Lease...52
4.4 Remote Events...53
4.4.1 The Nature of Remote Events..53
4.4.2 Design Issues 56
4.5 Transactions..64
4.5.1 The Nature of Transactions..65
4.5.2 Design Issues 67
5. DISCUSSION..71
5.1 Drawbacks 71
5.2 Future 74
6. REFERENCES 77

Abbreviations

2PC	Two-Phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
ACL	Access Control List / Agent Communication Language
ACM	Association for Computing Machinery, Inc.
AE	Application Environment
AP	Access Point
API	Application Program Interface
AWT	Abstract Windowing Toolkit
BT	Bluetooth
C/S	Client/Server
CLDC	Connected, Limited Device Configuration
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCOM	Distributed Component Object Model
DNS	Domain Name System
eJava	EmbeddedJava
EJB	Enterprise JavaBeans
GC	Garbage Collection
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HTTP	HyperText Transfer Protocol
IBM	International Business Machines Corporation
ID	Identification
IDL	Interface Definition Language
IEEE	The Institute of Electrical and Electronics Engineers, Inc.
IETF	The Internet Engineering Task Force
IIOP	Internet Inter-Orb Protocol
ILU	Inter-Language Unification
IMAP	Internet Mail Access Protocol
IP	Internet Protocol
ISL	Interface Specification Language
ISO	International Organization for Standardization

J2EE	Java 2 Platform, Enterprise Edition
J2ME	Java 2 Platform, Micro Edition
J2SE	Java 2 Platform, Standard Edition
JAR	Java Archive
JCP	Java Community Process
JDC	Java Developer Community
JFC	Java Foundation Classes
JIT	just-in-time
JMS	Java Messaging Service
JRE	Java Running Environment
JSR	Java Specification Request
JVM	Java Virtual Machine
KVM	Kilobyte Virtual Machine
LAN	Local Area Network
MIDP	Mobile Information Device Profile
MOM	Message-Oriented Middleware
MPI	Message Passing Interface
MTU	Maximum Transmission Unit
OMG	Object Management Group
OO	object-oriented
ORB	Object Request Broker
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
pJava	PersonalJava
QoS	Quality of Service
RAM	Random Access Memory
RMI	Remote Method Invocation
RMID	Remote Method Invocation Daemon
ROM	Read-Only Memory
RPC	Remote Procedure Call
RRMI	Reflective Remote Method Invocation
SCSL	Sun Community Source License

SDS	Service Discovery Service
SIG	Special Interest Group
SLP	Service Location Protocol
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TTL	time-to-live
UDP	User Datagram Protocol
UI	User Interface
UPnP	Universal Plug and Play
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WAN	Wide Area Network
WLAN	Wireless Local Area Network
XML	eXtensible Markup Language

1. INTRODUCTION

As companies and business in general become more and more global, the need for service distribution over the networks grows. There already exist a large number of various platforms and networks, including wireless environments that must be handled when transferring information between devices and applications. Also totally new areas of development are being introduced today. One of these very interesting areas is home communications, which means communication between domestic appliances and entertainment devices like television. These scenarios set new demands for distributed software development. Home communications is also one step more towards the *ubiquitous computing paradigm*, which means computing technology that recedes into the background of people's daily lives.

In addition, a greater degree of convergence can be seen between software components and hardware devices. They work together even more closely to provide not only applications but also services, which can be used by clients over different kinds of networks. The goals here are simplicity and connectivity, which means that very heterogeneous devices can be connected to work together without complex device driver installations and configurations.

The revolution of the Internet, and the networks in general, has been very fast and software industry has had difficulties to keep up. *Jini technology* [Sun99c] is one of the most important attempts to fulfill the high requirements of heterogeneous and fast changing network design and implementation issues. Based on the *Java programming language* [Gos96], Jini technology tries to deal especially with very complex network issues, like partial network failures and latency. Jini offers relatively lightweight solutions for these issues, which cannot be covered by the traditional distributed systems, at least with solutions that are lightweight enough.

The industry is moving from the disk centric desktop computing and distributed applications to a network centric service model. Jini fits into the picture very well, because in a Jini system, every involved entity is defined as a service,

which can be accessed through its *interface*. A service can mean various things e.g. a disk storage, a printer, or a search service.

Jini is a powerful framework for creating and deploying distributed services. Jini is based on the model of a *lookup service*, which represents a Jini system. All services, including the lookup service, are found by using *service discovery protocols*. In addition to these basic concepts, Jini includes three programming models, which are *leasing*, *remote events* and *transactions*.

Jini was originally developed to be used with pure hardware based embedded devices. The purpose was especially to connect heterogeneous devices with various capabilities together. However, it was soon discovered that Jini is also capable of pure software solutions. This proves the flexible nature of a well-designed architecture with object-based programming capabilities. Entities are communicating not straight through their implementations but through their interfaces.

A revolutionary feature of Jini is the ability to move objects freely over a network and this way provide spontaneous joining and leaving from the network. Another important issue is that the Java programming platform provides platform independence and code portability also to Jini systems. And because the Java programming language gains more popularity all the time, this helps also Jini to be spread with it.

The purpose of this thesis is to provide an introduction to Jini and the difficulties it overcomes. After a short briefing about Jini in Chapter 3, all the main programming models are discussed in more detail in Chapter 4, which also provides some code examples. In this chapter, Jini's influence on the difficulties of the distributed systems is also discussed when appropriate. Before taking a deeper look at the Jini architecture, some background knowledge and concepts of the distributed systems in general are presented in Chapter 2. This chapter hopefully helps to understand, how difficult the distributed software designing is, and what special requirements it sets to developers. Nevertheless, some solutions and models presented in Chapter 4 should provide some assistance and ideas for the reader to weather out the basic problems. And finally,

drawbacks of this technology are discussed and some assumptions about the future of distributed computing are made.

2. AN OVERVIEW OF DISTRIBUTED SYSTEMS

There is an increasing need for distributed software and the number of such systems has exploded in the last few years. Globalization and the growth of companies compel them to distribute and expand their operations and systems [Ste98]. This happens both globally in the Internet and more locally in *Local Area Networks* (LANs) and in *virtual private networks* (VPNs) [Gle00]. User mobility and *ubiquitous computing* [Wei96] bring out even more challenges to build genuinely distributed computing systems in the future [Cou94, Esl99]. The development of thin mobile clients, which are constricted, networked devices, and multiple distribution level *client/server* (C/S) architectures [Lew98] also demand new technology from communication *middleware* and component-centric system design [Ste98].

Building a distributed system is much more complicated than building a local system. In a distributed system, there are many aspects that have to be considered, unlike in building a local system, because distributed systems can fail in a number of ways the local ones cannot [Wel99]. Important concepts to deal with are *latency*, *memory access*, *partial failure* and *concurrency* [Wal94]. This chapter sheds these problems, introduces the traditional distributed systems *unified object model* and discusses the affect of Java programming language and object-orientation on these problems.

2.1 Anatomy and Features of Distributed Systems

Distributed systems are built, because information is easier to maintain locally and the relocation of large information sets can turn out to be too difficult to accomplish [Far98]. In some scale distributed software components can be used to enhance fault tolerance in the network, since in a well-designed system other components can provide the same services as the components that have crashed.

Distributed applications are built on top of the network protocols, e.g. *Transmission Control Protocol/Internet Protocol* (TCP/IP) [Pos81] and higher level services like secure connections and directory services. The applications can also use services offered by *operating system* (OS). At the application level,

object-based distributed software can be broken down into the following parts [Far98]:

- *Processes* have the access to the services of OS. Server machines are able to run multiple processes simultaneously.
- *Threads* run independently within a process. Some OSs allow one process to control multiple threads. Threads must be synchronized to perform cooperative tasks.
- *Objects* are runtime instances of classes, which are the basic components of the object-based application design.
- *Services* (agents) are software components, which represent and are responsible for one task within the application. Services are composed of one or more objects and threads. One example is a Jini lookup service.

In the environment where multiple components are interacting, they might be implemented by using different languages and platforms. One way to accomplish platform independence is to use specific object interface descriptions, which can be converted to *skeletons* on the server side, and *stubs* on the client side (Figure 2-1). In *Common Object Request Broker Architecture* (CORBA) [OMG99b] systems the interfaces are described with *Interface Definition Language* (IDL). Other possibilities in different environments are *Interface Specification Language* (ISL) from Xerox [Jan99], and *Component Object Model* (COM) language from Microsoft Corporation [Mic00b].

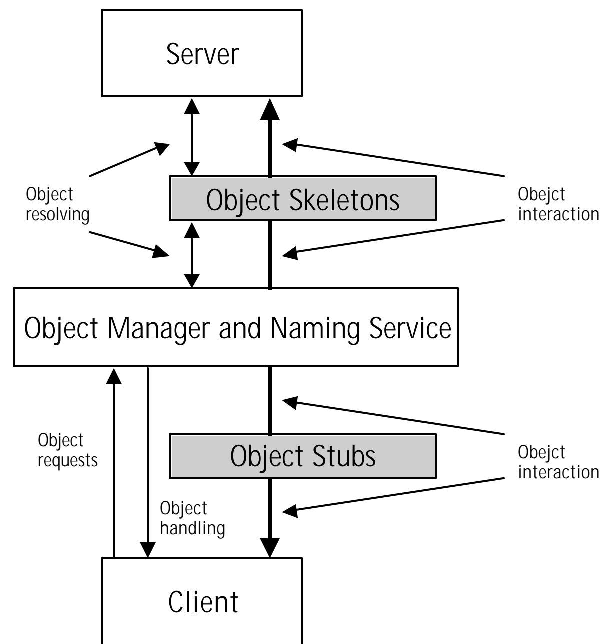

Figure 2-1. Remote object interaction through skeletons and stubs [Far98].

Object manager is the central entity of a distributed software environment. It provides references and mapping between clients and servers through skeletons and stubs. Object manager corresponds to *Object Request Broker* (ORB) in CORBA systems, and the *registry* service in *Remote Method Invocation* (RMI) [Sun99a] based Java systems. Other important tasks for object manager are dynamic object activation and deactivation, as well as storage provision for persistent objects.

Naming service supports the tasks of the object manager. Naming service is usually a registry-based service, which maintains up to date statistics about the current objects of the system. Clients use the naming service for resolving the names and types of objects, which they need to retrieve from the server. Requests are routed by the naming service.

2.2 Major Problems and the Model of Unified Objects

All the classical network systems have one thing in common: they try to make the network disappear from the programmers' point of view [Edw99, Wal94]. Still the relatively new remote object systems like CORBA and Microsoft Corporation's *Distributed Component Object Model* (DCOM) [Mic97] try to make the network transparent and simulate remote method calls as local by using

remote procedure calls (RPCs) and the concepts of *Message-Oriented Middleware* (MOM) [Lew98]. This has lead to problems concerning e.g. the correctness of programs, because the system can sometimes stay in an inconsistent state because of inevitable partial failures in network [Edw99, Wal94].

This vision of a transparent network along with the model of *unified* objects relies generally on the following principles [Wal94]: 1) The interface of an object is independent of the context in which that object is used, 2) failure and performance issues are tied to the implementation of the components, and should be left out of an initial design, and 3) there is a single natural *object oriented* (OO) design for components regardless of the context they will be used in. Along with these false principles, "The Seven Fallacies of Distributed Systems" is written by Peter Deutsch [Edw99]. These are:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology does not change.
- There is one administrator.
- Transport cost is zero.

These are exactly the issues Jini architecture tries to deal with by acknowledging them unlike the traditional distributed systems. It is obvious that in a distributed system occurs latency by courtesy of time reserved for data transfer. Invoking methods on a remote machine causes performance problems, even if the development of data transfer techniques has improved rapidly. The real *qualitative* problem occurs, because it is almost impossible to know, if the transfer of some remote component is just slow, or when the transfer has failed [Edw99].

In general, the performance of hardware has been slightly ahead of software demands in the past years, but this situation may change in the future, because of increasing requirements on bandwidth and new network services like *video*

on demand. Traditional distributed system design relies on the presupposition that the bandwidth of networks will increase in the future. That would make latency irrelevant or at least its impact would be reduced. But this is one of the matters that just cannot be predicted in advance. Although it would be possible to mask efficiency difference between remote and local method invocation, it requires new technology and changes to the programming models used [Wal94].

Even more important than reducing the impact of latency is to bargain for it. Because network latency also begets qualitative differences, not just *quantitative* ones, remote and local method calls cannot be treated as similar [Wal94]. This is why interface design of distributed systems is the most important thing and it must be tied closely with the architectural design of the system already in the initial phase of system design. This careful interface design makes real component code reuse easier, because the heavy burden of remoteness in implementations is transferred to interfaces. New object distribution frameworks like Jini rely on interfaces and lightweight class implementation, but many of the traditional systems suffer from these modifications that are needed to turn components from local to remote [CaD98].

Moreover, the underlying system must have a total control over the memory access to paper over full network transparency. This use of shared distributed memory leads to the situation where address space related pointers or object references cannot exist [Arn99b]. Because the use of this model requires a change in one's programming style, complete transparency between local and remote method calls is again given up, and it is not likely that one central entity can offer context for all kinds of applications [Wal94].

Even if the model of unified objects can be conceptually understood when dealing with latency, the situation gets more difficult, when considering partial network failures [Arn99b]. In local computing total failures are obviously easy to detect, because they affect all entities involved, and it might be possible to use the help of OS services. In network, even the use of reliable protocols offers no solution for detecting partial failures. This problem cannot be solved by using some library code or features of OS in a general manner to paper over the

difference between local and distributed computing, hence programmers must come up with a new application specific solution every time.

In bargaining for partial failures in a network, one of the most important things is, again, interface design, as the improvements made in the actual component code are not capable of dealing with the failures that inevitably occur. Because the state of the system cannot be known after the failure, communication interfaces must deal with *indeterminacy*. Two different things must be considered in designing distributed interfaces [Wal94]. One is to build interfaces, which are capable of perceiving the cause of failure, and the other is to provide interfaces, which enable reasonable state reconstruction after the failures.

When considering the model of unified objects between local and distributed objects, it is evident that the purpose of this model is to make distributed computing easier by treating it as it were local, e.g. in [Hic99] and [Hol98]. In the case of partial failures, there is only one approach to fulfill this model, and that is to make all method invocations remote, because localizing every call is not an option since distributed failure modes would otherwise be left unhandled [Arn99b]. Naturally this all-remote approach makes the whole OO programming more heavyweight, and forces local computing to be as complex as distributed, which of course is not the purpose.

Because in a distributed system there usually is no centralized entity, like there is an OS on the local machine, the concurrency and consistency failures can be devastating. The lack of a directive and supervisory entity entails that the failure of one component involved in the distributed computation can not be announced to the other components, which continue to function [Edw99]. The central problem is the consistency of the whole system after the failure, because there is no way of guaranteeing identification of the particular crashed entities [Wal94].

In distributed systems, there is a possibility that the entities must handle concurrent method invocations [Arn99b]. As to the unified objects model, the situation is similar when comparing concurrency issues and partial failures. All

entities should be semantically able to handle worst-case situations even locally, because ignorance of concurrent invocations would be catastrophic for the system. This indicates that the use of unified objects model is not possible in distributed object systems, but some other approach must be taken.

2.3 Demands of Distributed Systems

In general, end users are offered a much wider variety of services by distributed computing than by using single workstations. In addition to possible performance and reliability improvements provided by separated software components, the same hardware components and network resources can be shared across the network. This kind of functionality requires a lot from OS and other utility software.

One of the main challenges for system software, which is destinated to distributed environment, is *reconfigurability*. Scaling and upgrading some network elements and services within the system should not disrupt other elements and services [Bec99]. There are two kinds of timescales in distributed systems that should be addressed [Cou94]:

- *Short-term changes* like failed processes, network element replacement, load balancing, and data and process replication. These changes should always be able to be completed in runtime conditions.
- *Medium to long-term evolution*, like new role assignments to existing network elements and new element additions.

New demands for reconfigurability rise from the mobility of networked devices. These devices must be adaptable to local systems even in runtime conditions. Hence, the dynamic resource manager of the system must be able to handle even indirect resource requests, and provide configurations on the fly [KoP99].

The characteristic, which describes the extension level of the system, is called *openness*. A system can be open or closed with respect to both hardware and software extensions. The level of openness is defined by the degree to which new, shared resources can be added to the system without disrupting the

existing services [Cou94]. To attain openness within a distributed system, all the key interfaces must be published for the developers [Lew98]. This entails careful documentation processes and specifications. These processes often bypass the official standardization procedure, as it is usually too slow. This can undoubtedly lead to problems in compatibility issues.

Quality of Service (QoS) requirements are usually demanded by the end users of the services in the distributed system. In general, these measurable properties normally include at least performance and response times, reliability and availability of services, and security issues related to data transfer [Cou94]. These requirements are common for all distributed systems. In networks QoS properties are usually defined more accurately, and in a more network protocol and element related manner [Slo94]. General requirements presented here just seek to emphasize important matters related to the distributed systems.

Sharing services and resources over the network is the most important feature of distributed systems. Servers function as service managers and providers, and clients as resource consumers [Lew98]. This kind of C/S model (Figure 2-2) is the most widely used and best-known model for distributed systems [Cou94]. It basically uses a simple request-response protocol, where a client requests a resource, and a server responds to the request with the result [Com00b].

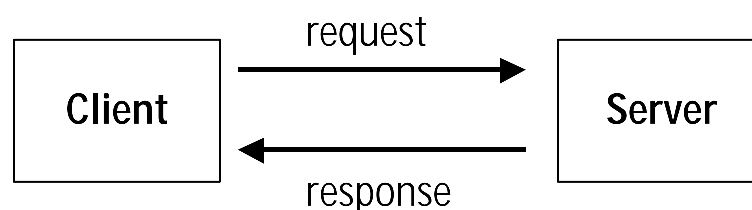

Figure 2-2. Simple client/server model.

The model of resource sharing can also be based purely on objects, in which each object represents a shared resource. This object-based model is more flexible and conceptually simpler than the C/S model, because all the objects and their references in the system are viewed in a uniform way [Fre99]. Also an OO technique called *polymorphism* enables closely related objects to react differently to the same events in the system through subclassing [Lew98]. In this

object-based model, object managers correspond closely to servers. Implementations of systems like this are more complex than in a C/S model, although the model itself is simpler. The greatest challenge is presented by the free migration of objects, because object managers must be able to access the state of the objects, also remotely.

When considering a multi-user environment, where multiple network elements and services are interacting, a great deal of communication actions takes place in parallel. Then, a resource sharing distributed system must deal with the resulting *concurrency* issues [Cou94]. There are generally two scenarios. The first is when multiple users are accessing the same application and invoking same methods simultaneously. This happens, because the distribution of services allows users and their applications to exploit multiple processors. The second is when multiple server processes, running at the same time are serving clients. Both of these scenarios require that concurrent access to shared resources must be synchronized.

One of the dominant themes in distributed system development is *scalability*. It refers to the ability of the system to meet future requirements. These requirements are usually related to growth of the system, which again means more users, more client computers, more service requests, and so on. Growing hardware update requirements are relatively straightforward to meet, but the real challenges are in software scalability [Cou94]. A handful of users have turned into hundreds of millions [Com00a], and distributed software should also be able to handle this new situation. Even if a lot of practical improvements emerge from the hardware development, software must be adaptable enough to be broadened into a larger collection of services from modular pieces of applications. Naturally, hardware development also sets new demands for software, for example because of multiprocessor systems and totally new pieces of incompatible hardware, like new *central processing units* (CPUs) [Bry99].

Fault tolerance does not mean that faults would not occur, but the system can cope with the situation where a fault has occurred, and the system should recover from it without losing any data [Cou94]. Two approaches exist as a

basis when developing fault tolerant computer systems. These are *hardware redundancy* and *software recovery*. Hardware redundancy refers to components, which are generally reliable and can back other failing components up, even in runtime. Hardware based fault tolerance is, at any rate, a very costly solution, because the system needs more hardware as a reserve. In distributed systems, the costs can be reduced by redirecting requests of clients to different servers, if a particular server fails. This solution is widely used in database-based environments, where data is replicated to multiple servers. This has similarities with *Internet Protocol* (IP) networks, where for example data packets are redirected through different routers if some of them are down.

Software based recovery is related to data storing. Systems should be designed and implemented in a way that make it possible to roll back any failed and incomplete actions without leaving any updated permanent data in inconsistent state. To achieve this ability within a system, a transaction based software model is often used. Naturally it makes software development much more complicated, but this approach gives many advantages when dealing with remote network failures [Edw99].

As transparency of a network is a fallacy from the developers' point of view, some levels of transparency are usually implemented for the end users of the distributed system to hide network complexity, and, consequently, to this way improve the usability of the system and *user interfaces* (UIs). Especially from the user point of view, several levels of transparency can be denoted [Cou94, Orf96]:

- *Access transparency* treats locally and remotely accessed objects identically.
- *Location transparency* enables accessing information in a similar way separately from the location of the information.
- *Concurrency transparency* means that multiple processes can access shared resources without interference.
- *Replication transparency* hides the difference between replicated data in the network.

- *Failure transparency* enables users to complete their tasks without the knowledge of some component failures.
- *Migration transparency* allows information to be moved to a new location without affecting to the operations of the clients.
- *Performance transparency* makes possible runtime reconfigurations according to the workload variation to improve the performance.
- *Scaling transparency* prevents rewriting of client software and system structure changes, when scaling the system up.

The first two transparencies are usually referred to together with the term *network transparency*, which is considered to be the most important case when utilizing distributed resources. It must be noticed that these transparencies are not considered from a developer's, but as an end user's point of view.

Security hazards in distributed computing are a very difficult and challenging topic. Security within clients, and on the other hand within servers alone, is not enough, because of the highly accessible nature of a network itself. Also compromising between high security and performance issues is a hard one. The role of network security is growing all the time, because more and more services are transferred into digital form, and to be used in a network like the Internet. Threats are multiform, and they include cases like information *leakage, tampering*, *resource stealing*, and just vandalism with *viruses* and *worms* [Cou94].

Several technical solutions are available for improving security [Orf96]. One of them is *user authentication*, which usually means that the system uses user names and passwords to identify users. Based on *Access Control Lists* (ACLs), the server grants rights to use resources to already authenticated users. This is called *authorization*. Administrative network monitoring is relatively easy, and it is usually used for user activity tracing, which means building up *audit trails* related to suspicious clients.

Also *encryption* and *cryptographic checksums* are used to increase security in networks. Both of the techniques are based on keys, which are used for coding and decoding messages, and exposing data changes during the communication

between distributed participants. Nevertheless, encryption in particular increases overhead in messages and can cause performance problems. Cryptography is also used to implement digital signature mechanisms. At any rate, authentication, authorization, and some level of encryption together usually form a minimal security solution for distributed system [Cou94, Far98].

2.4 Java's Affect on Difficulties

Java's designers were mainly concerned with reliability, simplicity and architecture neutrality when developing this OO language from its predecessor, *Oak*. When it was obvious that Java language would be used mainly to network programming, the demands on properties grew and the language was designed to support networking, security and multithreading. This makes Java a good solution for distributed programming [Far98, Sri97].

The minimum building block in OO programming is a class. This feature offers applications to be well defined and structural in the sense of designing and programming [Far98]. Because it is relatively easy to map natural entities and phenomena as objects [KoK96, Lew98], which are instances of the classes, the distribution of these objects is also simple and straightforward. With the help of Java's network support, the whole system can be build with the distribution issues incorporated from the ground up [Mah00].

When developing a distributed system, it is important to have the ability to separate the object's abstract interface from its implementation. By this ability of OO languages like Java, the implementation can be changed or even moved into the different location without changing the whole system, or rewriting client software. The critical elements of the Java *Application Programming Interface* (API) are based on this ability [Far98] and it makes it possible to develop platform independent implementations, for example components of *Abstract Windowing Toolkit* (AWT) on each platform, like Windows and Macintosh environments, by using abstract interfaces.

Java's RMI uses abstract interfaces to define local stubs and remote objects (Figure 2-3) in order to invoke methods on remote *Java Virtual Machines*

(JVMs) [Lin99] (Figure 2-4). This concept is also common in other distributed object systems such as CORBA, which uses IDL to define interfaces [Far98]. Even if RMI is a Java specific feature, and thus not compatible with other languages, Java supports IDL mapping, which enables interoperability [Lon99, OMG99a]. It must be noticed that even seamlessly compatible connected interfaces still do not guarantee correctness of the system, but a consistency checking using synchronization techniques must be made [Gen99].

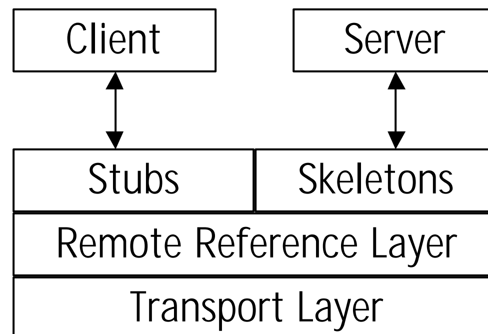

Figure 2-3. RMI system architecture [Pag97, Sri97].

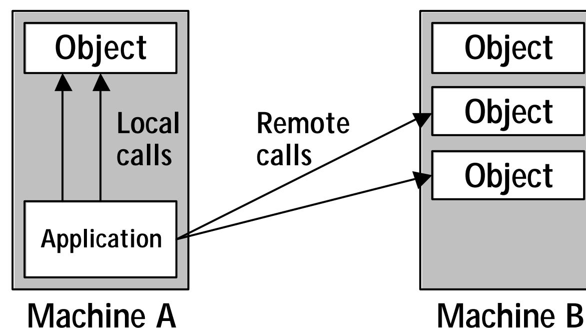

Figure 2-4. Local and remote method calls [Mah00].

There has been a lot of discussion about different kinds of distributed computing middleware solutions. The strengths of RMI are clearly its Java based security [Bus98], efficiency as a result of simplicity, and the ability to transfer *serialized* and *marshalled* data objects. Some solutions to make the performance of RMI even better exist, for example *Reflective RMI* (RRMI) [Thi98], and the use of better optimized object serialization mechanism [CaB99, Nes99]. If an existing distributed system with multiple programming languages is being developed further, the decision between CORBA and RMI points towards CORBA and its language independence [Kap99, WuD98], but in the case of a new system, the choice is easily pure Java approach with RMI.

Java code can be compiled into *bytecode*, which is platform independent and can be run on similar JVM on every supporting platform [Far98]. This makes it possible to develop Java based distributed systems on wide variety of different machine and operating system architectures. The drawback of Java has been, in spite of platform independence, performance. This matter has been improved, because many of the new JVMs are using *just-in-time* (JIT) compilers to optimize the translation of bytecode [Pis99].

In addition, Java is tailored to different sizes by default. These *Application Environments* (AEs), often referred as *Java platforms*, are separately focused to servers and desktop *Personal Computers* (PCs), *Personal Digital Assistants* (PDAs), mobile equipments, and finally to electronic cards, and to even smaller devices [Sun98]. In specific, these AEs are *PersonalJava* (pJava) [Sun99h], *EmbeddedJava* (eJava) [Sun99g] and *JavaCard* [Sun00d] in addition to Java AE running on a PC. From the developers' point of view, the Java programming environment is divided into three different categories according to the specific market areas [Lun00]. These environments, starting from the Java language version 2, are *Standard Edition* (J2SE), *Enterprise Edition* (J2EE), and *Micro Edition* (J2ME). For each edition of Java, there are many free programming tools, which makes Java even more popular, also among distributed system developers.

In Java, there can be application-defined and system-defined *exceptions* and *errors*, which can be thrown and caught by wrapping any potential exception causing code inside exception handling blocks. All Java methods can throw an exception. The caller of the method is responsible for dealing with the exceptions or it passes it up the *calling chain*. Because an exception is represented as an object, the programmer can implement his own exceptions, and so indicate abnormal exceptions, for example in distributed system [Far98].

Java API includes low-level sockets (Figure 2-5), and data communication protocols, which can be layered on top of sockets [Mah00]. Protocols can be connection-oriented like *Transmission Control Protocol* (TCP) [Pos81], or connectionless like *User Datagram Protocol* (UDP) [Pos80]. Data streams can

be filtered and preprocessed with several stream classes in a `java.io` packet. On top of this basic level communications, high-level network support can be built, such as distributed objects schemes, remote database connections and directory services [Far98]. Another important feature is *Garbage Collection* (GC), which enables automatic resource renewal, even remotely with the support of RMI [Abd98].

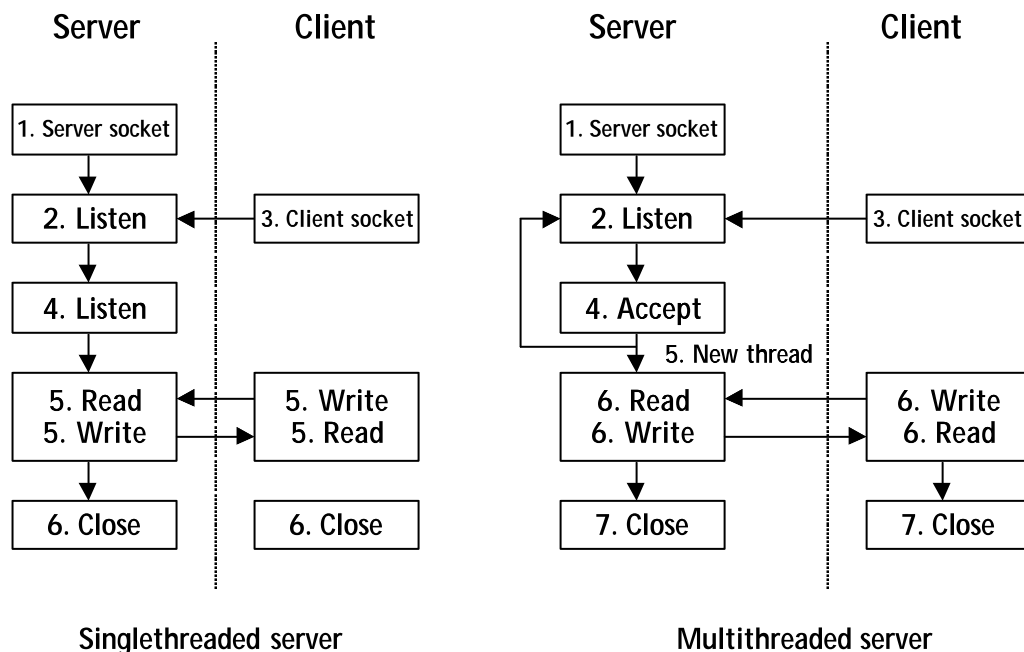

Figure 2-5. Client socket communication [Mah00].

Java has two dimensions of security for distributed systems: a secure local runtime environment and an ability to perform secure remote transactions. All local code is controlled by the JVM, which places restrictions on operations and capabilities of the code, downloaded from the network or not. For example, the downloaded code has no access to the file system, it can only make a network connection to the server it was originally downloaded from, it has no access to any local code or libraries outside the Java environment, and it has limited access to thread manipulation [Far98], so the code is said to be executed inside a *sandbox* [CIE99]. In the bytecode interpretation process, a JVM takes advantage of three different components, which offer built-in security for Java. These are, first, *class loader* for preventing built-in or extension class overriding, secondly, *class file verifier* for access and type restrictions observing, and,

finally, *security manager*, which keeps track of runtime access control and enforces restrictions, which are defined in *security policy* (Figure 2-6) [Pis99].

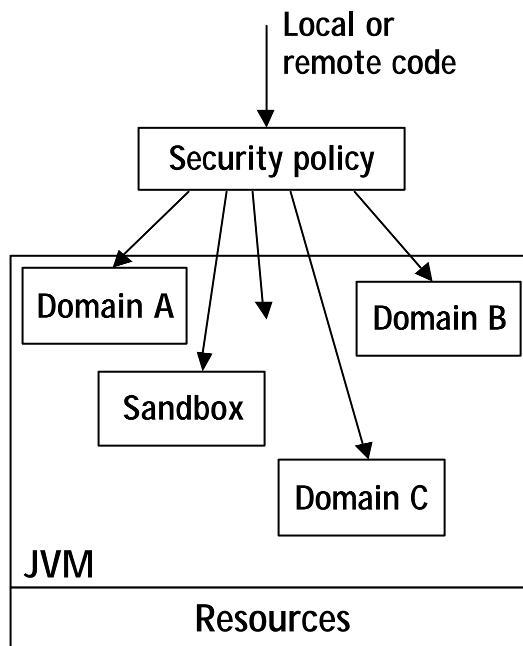

Figure 2-6. Access control mechanism in Java.

In remote cases, even if RMI itself does not support all crucial security features, Java supports an authentication system and encryption of data on remote and secure network connections with the help of classes in a `java.security` package.

Java provides *multithreading support* as a fundamental feature. This feature enables the use of multiple points of code execution in application. Every created class can extend the `java.lang.Thread` class and do its work within a separate thread. These threads can be grouped, prioritized, yielded to other threads, suspended or made to work together [Far98]. This kind of ability offers ways to build, for instance, distributed systems with synchronized parallel computing capabilities.

Every thread has its own *stack*, which stores local variables and method arguments (Figure 2-7). Individual stacks allow threads to modify their own local variables and call methods in the order they want [Mah00]. Still every thread

uses the same source code, the same static variables in *global data*, and the same *heap*, which holds new, allocated objects.

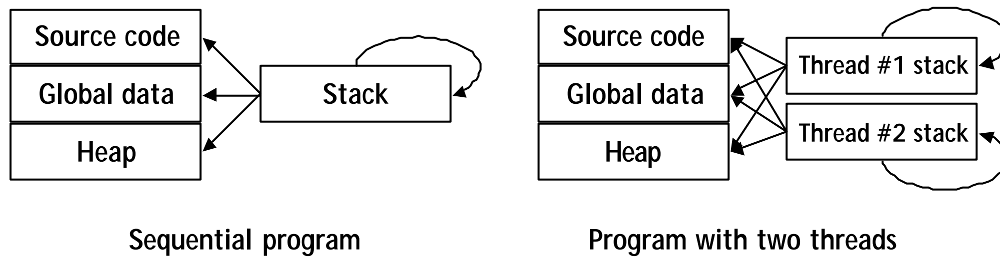

Figure 2-7. Threads in programs [Mah00].

3. WHAT IS JINI?

3.1 The Basic Idea

Jini technology (Figure 3-1) provides an infrastructure for defining, advertising and finding *services* in a network [Arn99a, Sun99c]. In practice, Jini is a set of protocols (Figure 3-2) and programming models for service provision using downloaded code and RMI [Cri99, Sun99a]. All the services are implemented using the Java programming language [Arn98]. The service is composed of defined *interface* and *implementation*. All the programming is done by using the OO way of programming. By using this paradigm and Java language, all the services are universal. This means that the execution of code is done exactly the same way on every platform hosting a JVM. This also means that the code of the service can be downloaded to the client when necessary. This feature provides a way for the clients to use services previously unknown to them. One of the main goals of this facility is that it allows spontaneous networking without any human intervention [Arn99a].

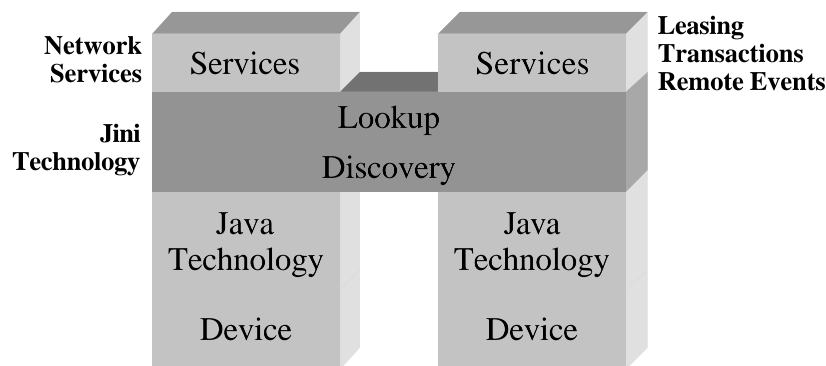

Figure 3-1. Jini connection technology.

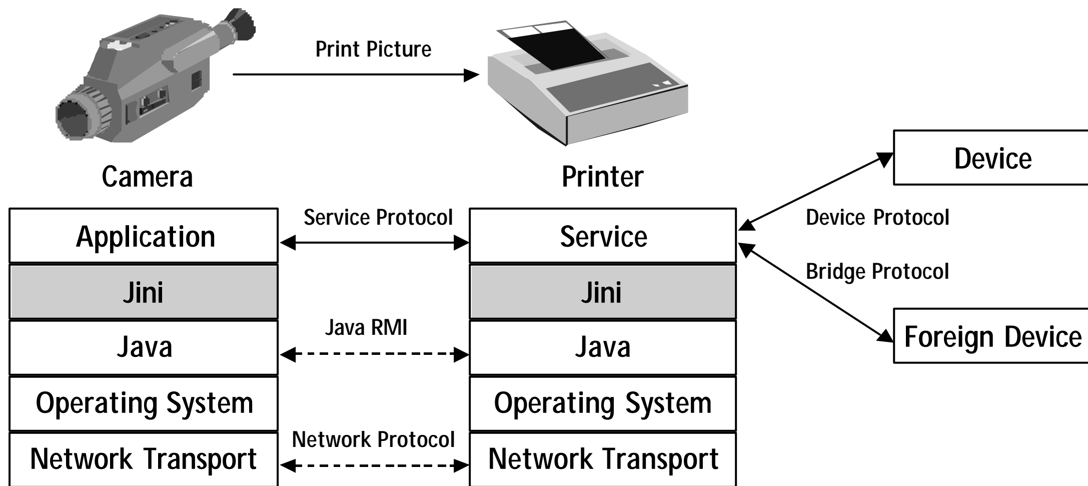

Figure 3-2. Various protocols within Jini technology architecture [CID99a, Sun99i].

The Jini lookup service represents the whole Jini *community*. It provides a place to advertise the services. The lookup service holds serialized *proxy* objects of the services. The type of the proxy object defines the type of service. Clients and services find the lookup service by using the *discovery protocol* and the services join to the community by using the *join protocol* [Arn99a, Sun99c].

3.2 Goals

The main goal of Jini architecture is to provide a type-safe and reliable way of providing services in the network. It is a framework for designing and implementing OO distributed systems [Wal99]. The aim is to allow services to be easily available and removable on a network to anyone who can reach them [Day00a].

Service deployment means plugging the service to the network, and it becomes visible and available, as easily as possible, to those who want to use it. The service can be pure software, hardware, or a combination of them. The idea of a flexible network is expanded, because all the services available are reachable to other services and clients, and removing the service is not any harder. The network can reorganize itself to be able to continue working after changes [Arn99a].

Jini technology changes the idea of network applications to the idea of a service-based network. Everything is a service. The goal is that this architecture allows any device with a processor, some memory and a network connection to offer and use services available. Jini contributes in implementing the services and making them available on the network. It is also possible to automate some implementation issues with Jini [Sun99b, Wal99].

Because of the Jini's new programming models, the developer is forced to acknowledge the difficulties of the distributed computing. Thus, this framework offers better capabilities to deal with the issues, which are cannot be handled by the traditional distributed systems.

3.3 Java Platform

Java provides a universal type system for Jini architecture, which means that all the object types are understood exactly in the same way on every platform hosting a JVM offering properties, on which Jini architecture relies. These are RMI, *homogeneity*, *single type system*, polymorphism, object serialization and marshalling, code downloading, *Java Archive* (JAR) files, safety and security policy [Arn99a].

Remote communication between objects that exist on different address spaces is possible using RMI. Homogeneity means that all the code downloaded by clients is executed the same way on every *virtual machine* (VM). This is resulted by the universal type system, which allows downloaded code to work both for local and remote computing. Code downloading is based on two features: Firstly, Java objects can be serialized into a transportable byte stream form [CaB99, Har97] and, secondly, during the serialization process objects can be marked with *codebase*. Codebase refers to the network address, which points to the actual location, from where remote code can be downloaded to the client when needed. Marked and serialized object is called a marshalled object. Packed class files and data with optional meta-information file, which allows certain packages to be sealed for restricting the use of included class files, are called JAR files [Cri99]. Client machine virus protection is provided by the JVM according to the security policy definitions [Arn99a].

3.4 Lookup

Lookup service is Jini's analogue to the traditional naming and directory service, although it has much more capability. The lookup service is the place where clients look for services they need. It is also the place where services advertise their existence. Advertising means that services store their serialized proxy objects in a lookup service after they have found the lookup service by using the discovery protocols [Arn99a].

When the service has registered itself with the lookup service, it is returned a *lease*. This is an amount of time, which tells to the service how long the registration is available. The service must renew the lease, if it wants to continue advertising itself. Leasing is the mechanism that maintains the freshness of the list of available services. The services which do not renew their leases are dropped out of the community [Arn99a]. Furthermore, leasing is the basis for measuring and possibly charging for the use of a service [Cri99].

To get a particular type of service, the client contacts the lookup service. They find the lookup service in the same way as the services find it – by discovery. When the client receives a proxy object of the lookup service, the client requests it to find one or several services needed, which match the *template* provided by the client. The lookup service finds the particular service by comparing the type rules of the template against all the service types currently registered. When the match is found, the lookup service returns the serialized proxy object of the service to the client. After that the client can deserialize the object and invoke its methods. In practice, invoking the method means that the proxy object sends a request by using RMI to the actual service. This mechanism provides that the client does not have to know the implementation of the services, just the interfaces [Arn99a]. In most cases the lookup service is no longer needed and the client communicates directly with the service [Cri99].

A Jini service can be stamped with attributes. These attributes are set by the system administrator or the service itself. If the attributes are set, the client can specify its service requests more accurately. The lookup service then finds the service needed by the type and, additionally, by matching attributes. The

attributes are used in a way that the client does not define them at all, or if defined, they have to match exactly [Arn99a]. This matching model is the same that is used in *JavaSpaces* [Fre99].

Lookup services can be part of one or more *groups*, which represent physical location or logical set of the services. Also, services can be registered with a particular group. These features help to arrange the network services into logical set of services [Arn99a] and they also help maintaining the information about the services, because the clients do not need to update the *Uniform Resource Locators* (URLs) of the services, when they change [Arn99b]. The group is actually a string, and an empty string means a *public* group, in which all the lookup services should belong.

3.5 Proxy

A proxy object represents a service item, which is actually an instance of a class that implements a certain Java type object [Arn99b]. When designing and programming in the OO way, the implementation is encapsulated behind the interface of the class [KoK96]. It frees the designer to focus on programming a good API rather than the network protocols. API level designing is even more flexible, when it is combined with code downloading [Arn99a], which means the transfer of proxy objects.

In traditional distributed systems like CORBA, the interface definitions, expressed usually with IDL, describe the methods that a remote server understands. This places strong requirements at the receiving end of the messages sent across the network. The proxy that implements the service in a Jini system can be as complicated as required. The ability to add a layer of the client-side code leaves the network protocol and wire protocol implementations to the individual programmers of each service, and enables Jini system designers to concentrate on good APIs. This means that only the APIs must be standardized, and programmers can increase functionality by adding their own methods to the proxies as long as the service supports the standardized methods [Arn99a].

3.6 Discovering Services

A discovery protocol is used to find lookup services in a network. In the Jini architecture, IP based network is supported. When a service is plugged in the network, it starts to send *multicast messages*, which lookup services listen to. If the particular lookup service manages a group the service wants to join, the lookup service sends its own URL-style address to the service, which then contacts the lookup service in that address using a *unicast join protocol* [Arn99a].

Registration is done using the unicast join protocol, which uses a URL-style location definition to get the host and port of the lookup service. After the connection has been established, the service downloads the proxy code of the lookup service and deserializes it [Arn99a]. The joining process is completed after the service has sent its own proxy object to the lookup service by calling the registration method of the lookup service proxy. This service item then functions as an advertisement of the service [Cri99].

Moreover, the lookup services send multicast messages into the network to advertise their existence. This implies that although a new service cannot send multicast messages, it can still register with the lookup service by responding to the multicast messages sent by the lookup service [Arn99a].

3.7 System Requirements

Although Jini is a very lightweight framework, if it is compared to the traditional programming solutions, it is still so new that it is not optimized for very limited devices yet, even if it has been the target in the first place. Hardware requirements for the devices running Jini capable services are still relatively high, because of the *Java Running Environment* (JRE) requirements. Despite the fact, the improvements in pJava and eJava AEs are going to the right direction. They both already have some level of support for RMI, in pJava as an optional feature, and in eJava as a configurable feature [Sun99g, Sun99h]. According to these specifications, the minimum hardware requirements of a Jini enable device related to the JRE can be defined as the following:

- Less than 512 kilobytes of *Read-Only Memory* (ROM).
- Less than 512 kilobytes of *Random Access Memory* (RAM). Basic Jini classes need approximately one hundred kilobytes, but RMI requires more than 250 kilobytes of memory to work [Sys00].
- 32-bit processor with at least 25 MHz CPU speed.

A basic Jini server, which hosts and runs a lookup service, must have at least two additional software components running [Per99] with working UDP and TCP enabled IP stack [Edw99]. First, a server software, which is capable of serving the clients with downloadable Java code. It is usually a *HyperText Transfer Protocol* (HTTP) server. Second component is an *RMI Daemon* (RMID), which makes remote method calls in practice possible to invoke [Aye99]. Naturally, the server must also host the JRE with a full JVM to support RMI. Optionally the server can run a transaction manager software, if transaction based activity is needed in the network. On the server side, these software components together form a basic Jini lookup service running environment.

4. BASIC CONCEPTS OF JINI

In this chapter, the basic concepts of Jini architecture are described. In practice, the basic concepts are implemented as software libraries and code conventions. Jini comprises five basic concepts that form the relatively simple core of this Java extension. Discovery and join protocols are used by entities to find services and bond with a Jini community. Discovery also allows spontaneous networking in practice. Lookup process controls the way the actual code is transferred between the clients and services and it works in a directory service role and beyond. This is the only basic process of Jini, which requires some user administration [Edw99]. Leasing allows Jini communities to be self-healing and restricts long-living services from growing too much. Remote events make it possible to announce other services the change of state of a particular service in the community. This feature has some similarity to the model of *Enterprise JavaBeans* (EJB) [Sun99e]. Transactions enable guarding against partial failures in the network and help in concurrency issues of Jini. They also help to maintain known states of services when multiple services are involved in the same computation. This contributes to the robustness of the network [Edw99].

4.1 Discovering and Joining Jini Communities

Discovering the services of the network is essential to maximize the potential of the networks, especially in wireless environments. At the same time numerous companies are developing mobile devices with network capabilities with high speed. Since the various handheld devices and portable computers are all highly mobile, their network administration and static network configuration become extremely complex, because of the large amount of networks, protocols and services.

There have been attempts to solve the problem of service discovery with different kinds of algorithms and network protocols. Sun Microsystems' Jini *discovery protocol* is not the only framework provided, but there are many other IP-based solution possibilities like the *Universal Plug and Play* (UPnP) discovery [Mic00a] from Microsoft Corporation, the *Service Location Protocol* (SLP) [Gut99a], the *Salutation* from Salutation Consortium [Sal00] etc.

When a new entity - an application or a service - wants to use the services of other Jini services, it must first discover available lookup services, which practically represent Jini communities in the network. To perform this search, the entity uses a discovery protocol to find the lookup services needed and then uses a *join protocol* to make its own services available to others in the community. In this way the entity becomes a part of the Jini community, can advertise its own services and use services provided by other members [Edw99].

An entity gets references to one or more lookup services by examining the return values of the discovery protocol. There are three kinds of protocols, which are associated with accomplishing the discovery process. These are the *multicast request protocol*, the *multicast announcement protocol* and the *unicast discovery protocol* [Edw99]. The use of a variety of protocols makes the Jini discovery process more flexible and adaptable to different kinds of network topologies [Day00a]. The first two protocols are serendipitous and make use of a *User Datagram Protocol/Internet Protocol* (UDP/IP) multicast, while the latter uses a TCP/IP unicast protocol facility [Edw99].

4.1.1 Discovering Lookup Services

A multicast request protocol is used when an entity needs to discover lookup services in a LAN after reconnection or for the first time. The components involved on the requesting side are a *multicast request client* and a *multicast response server* [Sun99c]. The multicast request client performs the multicast request itself (Figure 4-1). It requests references to the lookup services it is interested in. The multicast response server listens to the announcements of the lookup services. These two components always occur as a pair and never alone, but there can be many pairs in a single JVM [Edw99]. When the entity has performed discovery for some time, it stops and starts to listen to announcements from the lookup services [Arn99b].

On the lookup service side, there are a *multicast request server* and a *multicast response client* running. These two components join to form *a multicast request*

service. There is typically one service of this kind in a single JVM (Figure 4-1). A multicast request server listens to incoming multicast requests made by entities. A multicast response client responds to the requesting clients and sends them a proxy, which allows them to communicate with the lookup service [Arn99b]. The proxy object is actually an RMI stub, which functions as a reference to the discovered lookup service [Aye99]. However, it must be noticed that the multicast request service itself is not based on Java RMI, but it uses a UDP facility of the network *transport layer* of the *Open Systems Interconnection* (OSI) model [Cou94] from *International Organization for Standardization* (ISO).

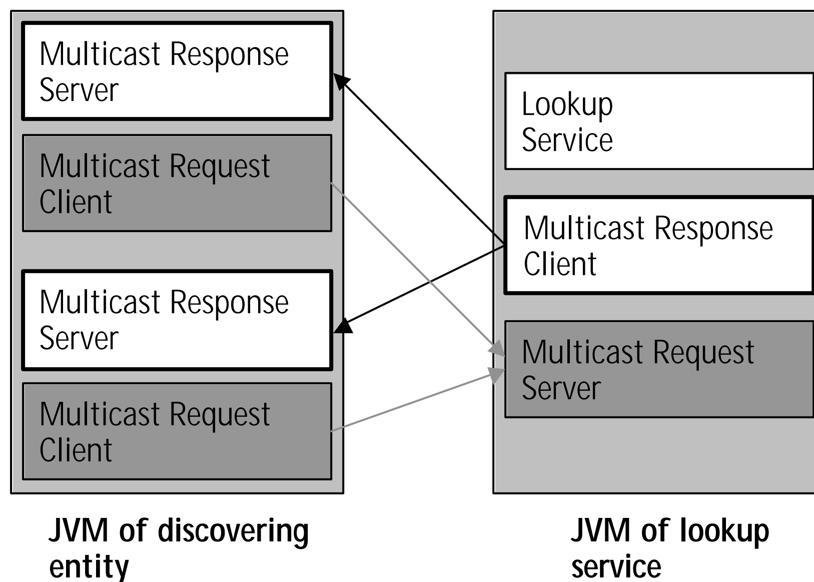

Figure 4-1. Multicast Request Protocol participants [Arn99b, Sun99c].

An entity performing the discovery first establishes a multicast request client. This component will send *a multicast discovery request packet* (Figure 4-2) to the multicast request service, which lives on the well-known network endpoint in LAN, usually in port 4160 [Aye99]. The entity also establishes a TCP server socket for listening to incoming connections. This TCP server is actually a multicast response server.

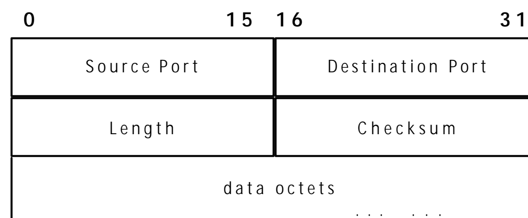

Figure 4-2. UDP header format [Pos80].

All the information (Table 4-1) that the entity sends within the datagram must fit into 512 bytes of data [Arn99b]. It is a size of the body of a single UDP datagram, which IP guarantees to arrive intact, if unguaranteed delivery is successful [Edw99], when the header size is also counted. This is because IP implementations are not required to handle bigger *maximum transmission units* (MTUs) [Mur98]. However, the information sent must not be truncated freely to meet this limit, but multiple multicast messages must be sent [Sun99c]. The data is represented as a marshalled object, which first has to be serialized, when it is written in an output stream as a byte array (Example 4-1).

Count	Serialized type	Description
1	Int	Protocol version (current value is 1)
1	Int	Port to connect
1	Int	Number of founded lookup services
variable	ServiceID	Founded lookup services as an array
1	Int	Number of groups the entity wants to discover
variable	String	Founded groups as an array

Table 4-1. Multicast request packet body [Sun99c].

```
int protoVersion;
...
java.io.ByteArrayOutputStream byteStr = new
  java.io.ByteArrayOutputStream();
java.io.DataOutputStream objStr = new
  java.io.DataOutputStream(byteStr);
// Writing all the data to the objStr.
objStr.writeInt(protoVersion);
...
```

```
Byte[] packetBody = byteStr.toByteArray(); // final product
```

Example 4-1. Generating a discovery request as a byte array to the output stream [Sun99c, Sun99f].

The entity must maintain a set of objects that contains *identifications* (IDs) of lookup services already found. Objects are instances of the `net.jini.core.lookup.ServiceID` class and they are added to the set, when the entity receives a response from the lookup service. The discovery requests are sent with periodic intervals. They contain connecting information for the entity's multicast response server and the latest set of founded lookup service IDs. The time interval between sent requests and the number of them is not defined, but the recommendations are five seconds and seven times [Sun99c]. The discovery process is finished, when the entity has received references to the lookup services it needed. If this does not happen, it must start using a multicast announcement protocol [Arn99b].

During the discovery process (Figure 4-3), a multicast request server binds a datagram socket to the well-known port, where the multicast request service resides. This service receives the incoming requests. It must respond to the requests, when the set of founded lookup service IDs is empty. It also has to react when its own ID is not in the set of IDs and some group requested matches some group this service belongs to as a member. Otherwise, it must not respond [Arn99b]. When the request must be responded to, the service connects to the requesting entity's multicast response server and provides it with a lookup service *registrar* (a proxy object) using the unicast discovery protocol.

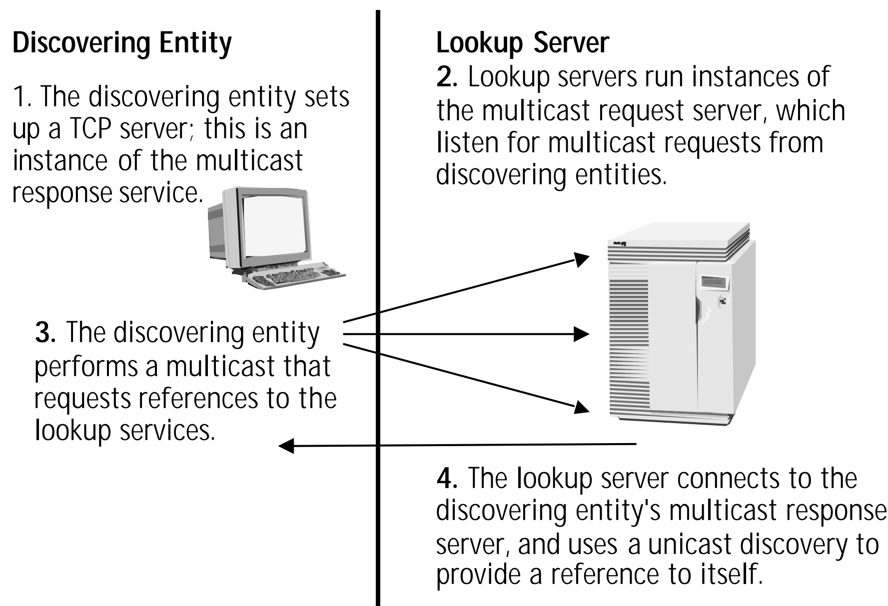

Figure 4-3. Multicast request protocol [Arn99b].

One thing to consider, when using multicast messages, is the situation called *a packet storm* [Edw99]. Because *well-behaving* Jini entities must be able to perform multicast discovery messaging when they have been started, there is a possibility that after a major network restart, the network is flooded with messages. One solution is to use Jini-provided discovery utility mechanisms [Sun99b] and especially a `LookupDiscovery` class, which provides initiation of multicast requesting with random time delays between 0 to 15 seconds [Sun99c] to scale concurrent request sending.

4.1.2 Announcing the Existence of the Lookup Services

A multicast announcement protocol (Figure 4-4) is used when a lookup service announces its presence to the members of the Jini community in LAN [Edw99]. Announcements are sent at regular intervals, which are not mandated, but the recommended interval is 120 seconds [Arn99b]. The announcement is sent by *a multicast announcement client*, which is a long-lived process residing on the same system as the announcing lookup service, and its *time-to-live* (TTL) is the same. The receiver of the announcement is *a multicast announcement server*, which lies on every entity, which listens for announcements [Arn99b].

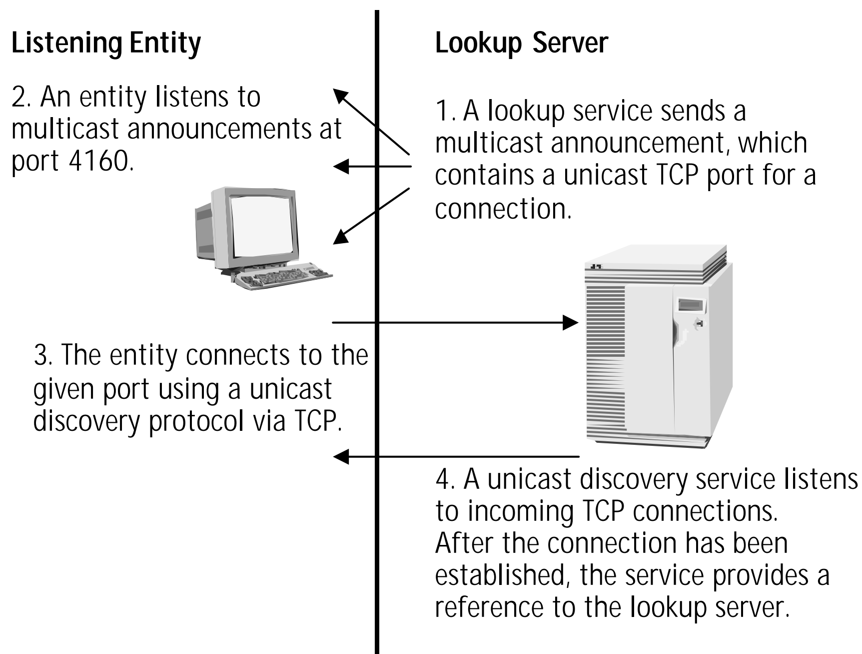

Figure 4-4. Multicast announcement protocol [Aye99].

The restrictions and limitations concerning the size of the UDP packet body (Table 4-2) are the same as in the multicast request protocol. The default well-known port for multicast announcements is also 4160.

Count	Serialized type	Description
1	Int	Protocol version (current value is 1)
1	String	Host name for unicast discovery
1	Int	Port number for unicast discovery
1	ServiceID	ID of the lookup service
1	Int	Number of groups in which the lookup service belongs
variable	String	Names of the groups in which the lookup service belongs

Table 4-2. Multicast announcement packet body [Arn99b].

The lookup service constructs a datagram socket object, which it sends to the well-known multicast endpoint where the multicast announcement service resides. The lookup service also establishes the server side of the *unicast discovery service* [Arn99b].

The entity establishes a set of service IDs, which it has heard of. This set is constructed from the set formed using the discovery request protocol during the discovery phase. It begins to listen to announcements by binding a datagram socket to the well-known multicast endpoint where multicast announcement service operates. The received announcement is ignored, if the ID of the advertised service is already on the entity's set of heard services or the entity is not a part of the group the announcement is meant for. Otherwise, the entity performs a unicast discovery to get a reference to the lookup service and ads its ID to the set of known services. The host and port needed to perform a unicast discovery, are retrieved from the announcement sent by the lookup service [Arn99b].

When implementing a lookup service and its multicast announcement protocol, the question of network traffic emerges. It is not efficient to have three separate calls to perform the existence announcement and send the proxy object. Still, this approach reduces the network traffic, compared to sending proxy objects straight into the network, which could be considered as an optional approach. There are two reasons for that. First, the size of the multicast datagram is limited to 512 bytes, which means that relatively large proxy object should be split to multiple packets. Second, most of the sent announcements are useless [Edw99], except when the lookup service is started for the first time or restarted after shutdown. Another issue is the TTL value of multicast messages, which can also flood the network in normal cases. The value is not mandated in a Jini specification, but the recommended value is 15 [Edw99].

4.1.3 Unicast Discovery Protocol

Although entities need to discover local lookup services, the end user may want to use other services outside the LAN. A unicast discovery protocol is used when an entity knows a particular lookup service and wants to connect directly to this. The entity knows the name of the lookup service, which is similar to the URL syntax with *jini*. For example `jini://cs.joensuu.fi` means lookup service running on host `cs.joensuu.fi` with the default port 4160. The unicast discovery protocol is also used when lookup services are connected to form a set of communities called a *federation* over a *Wide Area Network* (WAN)

[Edw99], and when lookup services initiate responses to multicast discovery requests [Arn99b].

Instead of a multicast UDP, the used protocol is a simple request-response protocol based on the unicast TCP [Sun99c]. When it is used as a part of multicast protocols (Figure 4-5), it works in the same way as in a stand-alone case (Figure 4-6). The difference between these two cases is that the information to perform a unicast discovery has been received from the data delivered by the former communication, but the stand-alone unicast discovery needs pre-configuration of the location (host name and port number) of the lookup service.

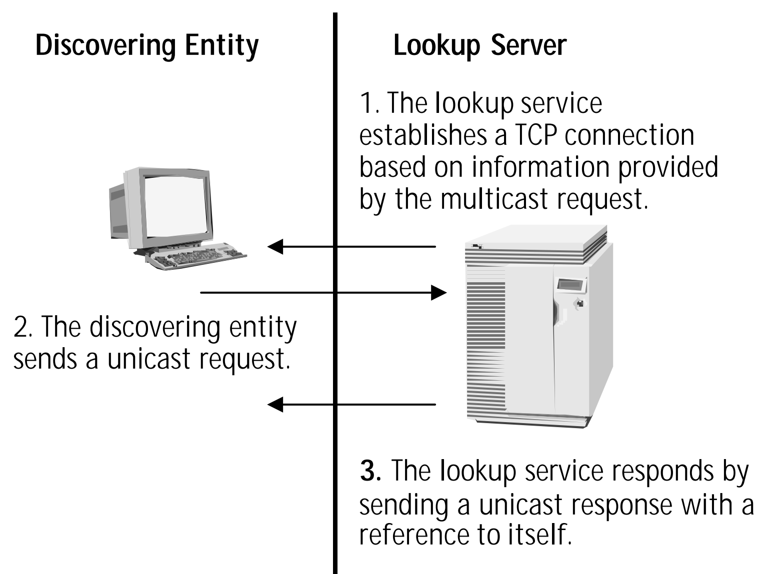

Figure 4-5. Unicast discovery initiated by the lookup service [Arn99b].

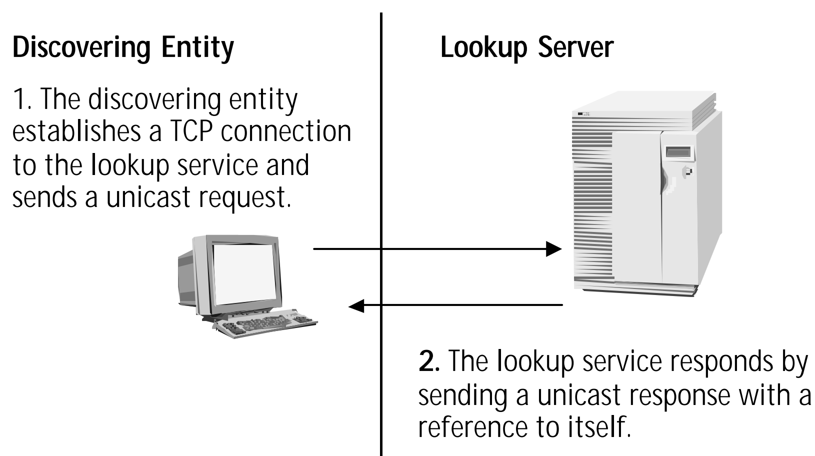

Figure 4-6. Unicast discovery initiated by the discovering entity [Sun99c].

The class that provides an interface for performing the unicast discovery is the `net.jini.core.discovery.LookupLocator`. It is a `Serializable` class, in which method `getRegistrar()` returns the proxy object of the lookup service discovered as an instance of the class `net.jini.core.lookup.ServiceRegistrar` [Sun99c].

Another consideration in all the discovery protocols is that there are no Jini-provided authentication security features included in the Jini release 1.0 [Cri99, Edw99]. Sun Microsystems has promised to provide this within the future releases. In practice this means that there should be a possibility to sign and certify class files in the Jini class loader and also to have a new version of RMI, which allows marshalled objects to be signed and certified as well [Cri99]. Moreover, security policies should be applied to downloadable objects rather than to classes themselves.

4.1.4 Join Protocol

A join protocol includes the sequence of steps, which should be performed by services when they are starting up and registering themselves with a lookup service to become a part of a Jini community [Arn99b]. These steps include maintaining the persistent state of the service itself, a lookup service discovery and registration, leasing, updating the service, and group joining and leaving [Sun99c].

Persistent data is a complex thing to handle in distributed transient object environments, especially when data and devices are highly mobile. One particular problem is related to databases and file systems, because they still do not support object type data widely [Jah98]. Even if object databases are developed, they are too massive to fit into the memory or a disk of mobile devices. In Jini this can be solved with a separate storage service, which saves persistent data on behalf of some service, but this still does not solve the original data type problem completely.

A Jini service should store some information (Table 4-3) about itself to be able to recover from restarts and crashes. This information is mandatory according to the Jini specification [Sun99c].

Item	**Description**
Service ID	A service ID assigned by the lookup service, when the service is started for the first time.
Attribute set	Attributes that describe the lookup service entry of the service.
Group set	Groups that the service wants to participate. Usually an empty string, which means a public group.
Lookup service set	Specific lookup services to register with.

Table 4-3. Persistent state of a service [Arn99b, Sun99c].

The service ID of the Jini service is 128 bits long number [Edw99]. It is very important that the service gets the unique ID in the very first contact with the lookup service, and saves this number in its persistent state. This is one way of providing Jini-style networking, because configurations of the services can be discovered and used after reconnections according to the IDs. Another issue is avoiding the serial number type of approach and the related centralized control of global registry of IDs. The service ID is calculated by the constructor of the `JoinManager` class of the lookup service using the random system clock based algorithm proposed by [Lea98]. These IDs are guaranteed to be globally unique until the year 3400 [Edw99].

To join the Jini community as a full member, the service must be able to perform the lookup service discovery using all or at least some of the discovery protocols. The minimum requirement is to be able to listen and react to incoming multicast announcements sent by lookup services. There is no specification in which order the discovery steps should be taken, but if the service is associated with specific lookup services, it should try to register with them first using a unicast discovery [Sun99c]. In addition to these requirements,

the service has to register with the same service attribute set and a single service ID with each lookup service [Arn99b].

Because all the connections between services and lookup services in Jini community are leased, the service must renew its leases to stay available to the community. If some communication problems occur, the action the service takes depends on the relationship with the lookup service [Sun99c]. If the particular lookup service has been discovered by using the multicast discovery, it is safe to discard the connection. The situation is different if the lookup service reference is in the list of persistent lookup services. Then the service is supposed to reregister with the lookup service. If the service is, for some reason, told to unregister with a particular lookup service, it should remove that lookup service entry from its persistent state set and cancel all leases associated with the lookup service [Sun99c]. The situation is reverse, when the service is told to register with a specific lookup service. Also the change of service attributes leads to changing these attributes with all registered lookup services.

Joining a specific group is a relatively easy task. The group name must be included in the set of persistent group list of the service, and the service should be able to perform multicast discovery to these groups. Leaving a group needs additional steps [Arn99b]:

- The service should remove the group from its persistent state.
- The service should remove all the lookup services, which match only to that group, and are discovered using multicast discovery.
- The service should continue performing multicast discovery with the reduced set of groups, or stop if the set has been reduced to empty.

The real implementation of the join protocol does not really exist. The protocol actually defines the steps and procedures to become a member of a Jini community and maintain the membership. The implementation is application specific like in almost every Jini protocols and procedures, but the specifications set the limits and minimum requirements as mentioned above.

4.2 Lookup Service

In distributed systems there usually is a central component that provides access to the distributed resources, because every client component in a distributed system may not have the tools to find services by themselves.

The lookup service (Figure 4-7) is the fundamental part of the Jini architecture core. It is the registry where all the available services advertise their existence, and where the clients go to find the services they need [Arn99b]. Jini's lookup service has some analogy to the concept of long-running name server, but the semantics is much richer, because even individual types of objects can be searched on the basis of the strong typing of the Java language [Edw99].

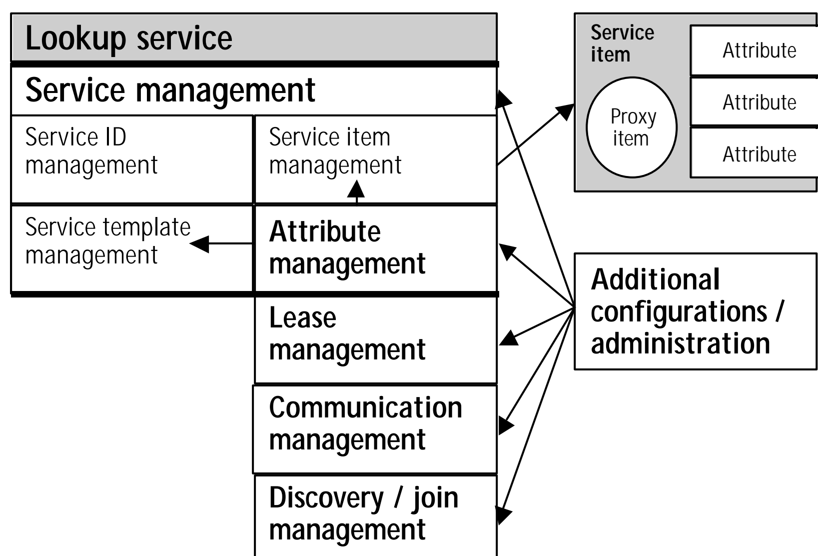

Figure 4-7. Lookup service architecture.

4.2.1 The Use of Lookup Service

Each service is presented as a service item, which is used to communicate with the service by downloading it, and a list of attributes, which describe the service characteristics, or possibly alternate interfaces. When services discover the lookup service, they upload their proxy objects, which are instances of the class `ServiceRegistrar`. These instances are used conceptually as a front-end interface [Edw99]. The `register()` method is used to join as a member of a community [Sun99d], and this registration is leased. Thus, the freshness of the

services available is guaranteed. As an argument of the method the service passes its own service item and attributes.

Another approach to examine the lookup service is from the client side, which naturally can also act as a service at the same time. Clients use the `lookup()` method offered by the `ServiceRegistrar` class [Sun99d], which, again, is not a real remote interface but a proxy, to find network services. To define the kind of service being searched, the client passes arguments for the method. Possible arguments are object types meaning the types of service items (Example 4-2), certain attributes, and defined lookup *entries* [Arn99b, Edw99, Sun99d], where an entry is an instance of the class, which correctly provides a `net.jini.core.entry.Entry` interface for accessing the collections of objects [Sun99c].

```
Object service; // returned service match
Object[] sTSet = [1]; // searched service types
ServiceTemplate tmpl; // searched service
...
serviceTypeSet = sTSet.newInstance[Printer, 1];
// constructing the template (ignore ID and attributes)
tmpl = new ServiceTemplate(null, serviceTypeSet, null);
// getting the matching service
// the name of the registered lookup service is 'foo'
try {
  service = foo.lookup(tmpl);
}
catch (RemoteException exp) {
  // the lookup service can not be reached
}
```

Example 4-2. Getting the matching service from the lookup service.

If the `lookup()` method returns a `null` value, there are no matching services [Sun99d]. In the case of multiple matches, a random instance is returned. It is possible to use two-parameter form of this method for defining the number of returned instances, but the exception handling can become very complex,

because the final success of the method must be discovered by examining the return values. A better solution is to use the `getEntryClasses()` method to receive a collection of services of same type, e.g. all available printers in a LAN.

Downloadable proxies are one of the key features of Jini architecture. When a client has downloaded a service item, it can use the service through the interface provided without having any formal information about the service, the communication protocol used, or driver configurations that are associated with it. The analogy to Java *applets* is obvious in some ways; applets can be downloaded and they do not require any administration. Still, they usually use *Graphical User Interfaces* (GUIs) and are meant for human-computer interaction. Downloadable proxies act more like agents; they are found, downloaded and used programmatically [Edw99]. It must be noticed that at this point there is a real-world problem associated with this kind of communication. To accomplish such a flexibility and spontaneity, the service interfaces must be standardized globally. This is a huge task to complete, because of the large amount of devices, attributes and vendors.

The attributes of the service item are represented as a set of attribute sets. An individual attribute set is actually an instance of a class, whose public fields represent each attribute. A service item can contain multiple instances of these classes and provide, for example, various implementations coded with different languages [Arn99b].

By using the event mechanism of Jini, the lookup services can be expanded to be more efficient producers of community services. Possible use scenarios could include announcing registered clients in certain matters like service unregistration, new service registration and service drop out [Sun99c]. This could be the way to keep up the consistency of the community. This is also helpful for the administration, because it can be informed about state changes and service problems, for example through an additional administration applet of the lookup service.

4.2.2 Design Issues

When a new service registers for the first time with the lookup service, it must be assigned a unique service ID. This is a feature that must be included in all lookup services [Arn99b]. If the service has no ID number when registering (Example 4-3), the assumption is that it is a new service. The lookup service still checks if the service equals with some service already registered, because the lack of an ID number may arise from restarting of the service already registered before. If the match is found, the new service replaces the old one, even if the attribute values do not exactly match The old service is removed, and all its leases are cancelled. The service ID of the old service is assigned to the new one, because it is safe to assume that service items represent the same service. Otherwise the ID number is normaly generated and assigned.

```
if (item.serviceID == null) {
  // the service has no ID number assigned
  int done = 0;
  int i = 0;
  int j = serviceItemList.length;
  if (j > 0) {
    while (i <= j && !done) {
      if (item.service.equals(serviceItemList[i]) {
        // replace serviceItemList[i] with item.service
        // assign the ID to the new service
        done = 1;
      }
      else i++;
    }
    if (!done) {
      // assign a new service ID for the service
    }
  }
  else {
    // assign a new service ID for the service
  }
}
```

Example 4-3. Checking the need for a new service ID of the registering service.

In discussing lookup service, some issues related to proxy objects should also be considered. There are actually four common cases, in which the use of proxy objects is different. These are [Edw99]:

- The service, which implements the whole service by itself. In this connection, the term proxy is somewhat misleading, because no other resources are needed. These kinds of services are usually pure software solutions.
- Downloaded object is an RMI stub, which allows the communication with the real service implementation. The service itself is usually some centralized service, like an *Internet Mail Access Protocol* (IMAP) based mail service, which allows Jini based applications access, for example, to the new mails in some folder without really distributing the mail service implementation.
- Attaching Jini based clients to legacy applications, which need some specific protocol for communication. The proxy includes an interface, which allows pure Java binding with the back-end protocol implementation.
- Some hardware component provides a service, which needs its own special protocol. Communication is accomplished through a proxy object, which acts as a device driver with Java interface adaptable for the application.

The service items within the lookup service can be arranged in a hierarchical order according to the types and attributes, although the collection is flat by default. The lookup service provides some methods to explore and browse the collection of services. This is useful for both the administrators and the clients [Arn99b].

When implementing the services, there are some standard attributes, which must be included inside the attribute set of the service. This requirement still needs to be worked on. The standard service attributes so far agreed upon are presented in Table 4-4. It must be made impossible for clients to change the

attributes through the service item globally [Edw99]. Sophisticated services and lookup services should provide own APIs for this purpose, because otherwise the services cannot fulfill the requirement of state change notification set by the join protocol.

Attribute	Purpose
Address	Geographical location of the service (country, city, street etc.)
Comment	Free-form `String` for comments
Location	Location of the service inside the organization (building, floor, room etc.)
Name	Human-readable name for service
ServiceInfo	Generic information (manufacturer, model, serial number etc.)
ServiceType	Service description (localized name, short description, icon etc.)
Status	Current operating status of the service (normal, error etc.)

Table 4-4. Jini standard service attributes [Edw99].

Another implementation issue is that the proxy object of the service is not leased by the client. Leasing is a framework to protect the system as a whole, not to protect any particular application, and the common leasing approach may not provide good solutions for individual clients anyway [Edw99]. Pragmatically it is difficult to say, which one, the client or the service, is the grantor of the lease, and which one is the consumer from the leasing point of view (Table 4-5), even if it is conceptually easy. Hence, the client can use the proxy as long as necessary, and after it has performed its task, it can be discarded. However, this impacts the implementation of the client, because it must notice whether the service back-end process is alive or not. This is performed by using careful exception catching and return value analysis after remote calls have been made. An additional mechanism to investigate the status of the service could be the implementation, which listens to incoming event notifications, e.g. service drop-out, from the lookup service the proxy was downloaded from, concerning the appropriate service.

Lease grantor	Lease consumer	Situation / problem
Service	Client	Client should renew the lease, but this technique still does not provide additional information about the crash of the service.
Client	Service	The user of the service must implement the mechanisms for lease management, even if it provides the running environment and resources for the service.

Table 4-5. Comparison between two different leasing possibility / problem [Edw99].

To increase the reliability of the Jini community, there are two options as to lookup services. The one is to have multiple lookup services to serve one community, and the other is federating the lookup services together [Edw99]. The first is actually a complex task, because of synchronization between the actual lookup service and its back-up service, and could also be a little oversized, at least for small communities. The other approach is a very interesting mechanism, because it provides a way to mirror company structure, administration boundaries, and hierarchically connect Jini communities (Figure 4-8), which are well capable of functioning autonomously in spite of partial network crashes. This federation mechanism is a great advantage in administration of the services compared to administration of one huge community, in which network latency and critical points of failure exist, and every service, not only the lookup services, has to have a company wide URL.

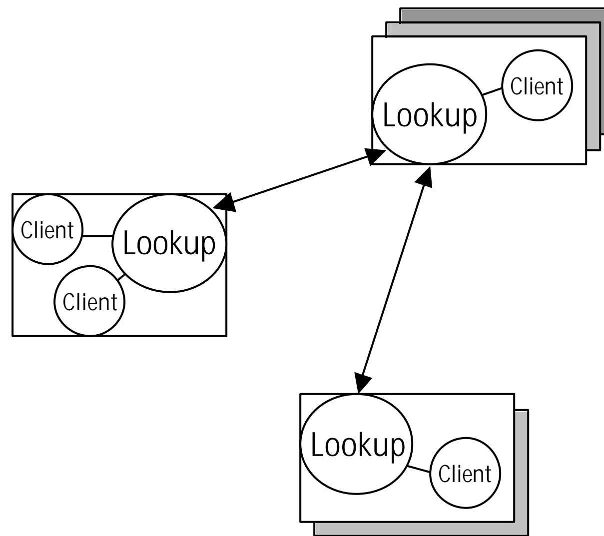

Figure 4-8. Connecting different Jini communities with lookup services.

4.3 Leasing

One of the desired properties of Jini architecture based distributed system is reliability. The problem is that in distributed systems failures are inevitable. They occur more easily and with greater complexity than in local systems. Leasing is a part of the Jini programming model, which helps programmers to bargain for these failures, which are going to occur no matter how reliable the real software itself is [Edw99].

The basic idea of leasing is conceptually very simple [Arn99b]: An object wants to use a particular resource, which another object offers. The former requests the resource and the latter grants a lease for the requested resource for some negotiated period of time. The lease must be renewed or it expires. If the holder does not renew the lease, it means that either it has crashed, there is a communication error or it does not want to use the leased resource anymore.

Characteristics of a lease [Arn99b, Sun99c]:

- A lease is a time period during which the grantor of the lease insures that the holder can access the leased resource. The time period can be decided completely by the grantor or it is negotiated with the client. It must be noticed that the guarantee to use any particular resource is gained through the implementation of the grantor.

- The holder of a lease can cancel it at any time during the period of lease. Used resources are freed.
- The holder can renew the lease by requesting it again during the lease period. The time period is negotiated in the same way as before. The grantor can reject the renewal.
- A lease can expire. The difference between expiration and cancellation is that expiration does not require any communication between the involved objects.

4.3.1 Benefits of Leasing

When a partial failure (Figure 4-9) occurs in the network, it is impossible to say, if one (or both) of the cooperating components of the distributed system has either crashed or the communication between them is down. With this time-based leasing concept, network failures can be detected and the used resources can be freed. The maximum time the failure can remain undetected is the duration of the lease [Edw99]. Without leasing, the resource provider can not free the resource in the case of a failure, because it does not know if the client is going to be back online or not. In the Jini system, a lease that is not renewed expires and the holder is considered not to be back online.

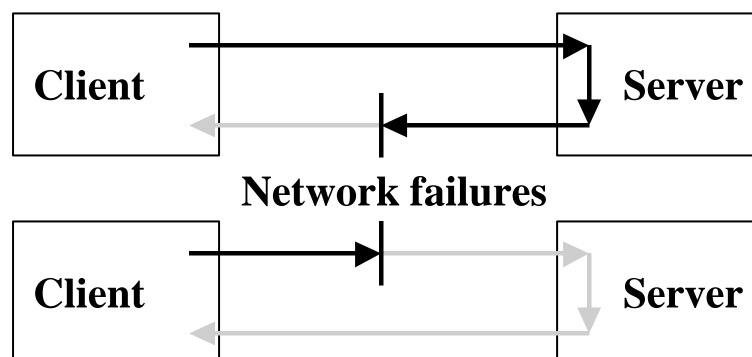

Figure 4-9. Two examples of partial network failure.

Programming with leasing is a relatively complex task, because both the grantor and the holder must be programmed to allow the concept of leasing in practice. Nevertheless, it provides real benefits to distributed system programming. In addition to the ability to detect failures in the network, the leasing model also provides a basic structure failed components to be cleared up automatically

[Edw99]. Because many of the large distributed systems are designed to be long-lived, it is impossible to administer them by hand. For example, the Internet or a global enterprise network can be considered as systems, which cannot be completely shut down for administrative operations or clean-ups, either from one physical location or from many locations exactly simultaneously.

The ability to clean up crashed components is autonomous, as the grantor of the lease knows when the lease expires. Since there is no central leasing control component, a distributed system with a working Jini leasing system can be self-healing [Edw99]. Naturally, the holder is not able to free resources after a crash, but the grantor can, because it has total control over the lease relationship. Consequently, it is guaranteed that dated data cannot be used by other members of the network, and the resources are freed automatically.

Another administrational benefit from leasing is that developing the components in distributed systems can be made online. When a component must be upgraded, all its leases are first forced to expire. Then the implementation can be changed without the danger of duplicate versions in network [Edw99], because the former version of the component has been completely dropped out from the community.

4.3.2 Design Issues

In distributed systems, there can be various devices connected with each other, located possibly in different time zones. This is why the resource leasing duration should be defined in *relative* time, not in *absolute* time, although the absolute time is more natural to some applications. The devices can have different representations of time and, further, a designer cannot know, how well the other devices are maintained. Even if the internal clocks of the machines are wrong because of keeping inaccurate time, the error is still crucially smaller when using relative time instead of absolute time [Edw99]. In fact, the actual expiration time of the lease is counted in absolute time because it is convenient for the programmers. A method called `getExpiration()` returns the absolute lease expiration time in milliseconds, counted from the start of "Java time", 1st of January 1970 [Sun99d]. This is why the necessary computations are easy to

accomplish by checking the current clock against the expiration time of the lease (Example 4-4).

```
// This method returns the time left before the lease
expires.
public long getRemainingDuration(Lease lease) throws
         UnknownLeaseException {
  return lease.getExpiration() - currentTimeMillis();
}
```

Example 4-4. Checking the lease expiration time against the current clock.

Another thing that must be taken into account when deciding the duration of leases is the latency that occurs in networks. This inevitable feature of networks sets a minimum bound for leasing time. This means in practice that the lease renewal request made by a client cannot be guaranteed to arrive at any fixed time. Thus, the duration of the lease must be long enough to encompass its renewal with defined probability, even if latency occurs.

The most obvious thing to consider when designing a leasing-based system is the lease negotiation process. In Jini, the negotiation process is kept simple as it minimizes remote procedure calls. Actually, there are only two procedure calls that must be executed [Edw99]. First the client requests a lease from the resource provider and proposes the duration. Then the grantor replies with the actual duration of the lease. Possible return values are 1) the requested time, 2) zero, which means that the permission to use the requested resource is not permitted, or 3) any value between these two. This approach is not as flexible and accurate as *multiround* leasing model [Edw99], but its benefits are obvious, because of its lightweight nature.

There is a further design issue to mention. Especially in embedded systems, the computing power and communication skills of small devices are limited. When these devices being involved in the Jini system, leasing can be done by a third-party object, which has the capacity to accomplish needed procedures [Edw99]. Now the actual client can use all its power to utilize resources more efficiently. Unfortunately, this can cause problems, because the actual consumer of the

resource is not responsible of the lease renewing any more. One type of situation is created by a crash of the service consumer. If the leasing service has been asked to lease some service for a particular time period, it will do it even if the consumer cannot make use of the resource any longer. Another scenario is that the leasing service becomes unavailable, and the consumer of the service does not know that the leases have expired. Both of these cases weaken the robustness of the community. Consequently, the option of a third-party lease manager has to be considered very carefully. Usually this approach fits to very reliable pure hardware devices, which are not capable of running leasing related software.

Although the concept of leasing may sound simple, it makes the whole programming model of Jini system more heavyweight. The leaseholder must show continuous interest in leased resources to keep them available by renewing the leases as shown in Example 4-5.

```
long duration = asked_lease_time; // in milliseconds
try {
  lease.renew(duration);
} catch (LeaseDeniedException lde) {
  // the grantor is unable or unwilling to renew the lease
  // the lease remains untouched
} catch (UnknownLeaseException ule) {
  // the grantor does not recognize the lease
} catch (RemoteException re) {
  // possibly thrown if the grantor is a remote object
}
```

Example 4-5. Renewing the lease using core method `renew()` and catching all thrown exceptions.

Instead of using the core methods of leasing, programmers can take advantage of higher level APIs. One of the most important classes is `LeaseRenewalManager`, which takes care of the lease renewal on behalf of the consumer [Edw99]. With this class it is possible to cancel, add and renew

leases with a single method call. This makes lease management much easier than the use of raw core methods.

Nevertheless, the resource provider must constantly keep an eye on resources to be able to free them when leases expire. It must also, naturally, have an implementation for lease duration allocation and negotiation as well as lease renewal mechanisms.

4.3.3 Managing a Lease

The `net.jini.core.lease.Lease` interface defines an object, which is returned to the holder by the grantor when the lease is given. The actual implementation of the object is hidden from the holder [Sun99c]. The object returned to the holder includes this interface and it is used for managing the lease [Edw99]. The reason why this is not a class but an interface is that the grantor can decide how it will communicate with the holder. Individual grantors can use various independent mechanisms for communicating, for example RMI, CORBA or raw TCP/IP-sockets.

Because the holder has to renew the lease, it needs tools for that. The interface of the proxy item of the lease provides a standard way of communicating with the grantor (Figure 4-10). Some of the method calls are local, e.g. `getExpiration()`, and some remote, e.g. cancellation and renewal of the lease.

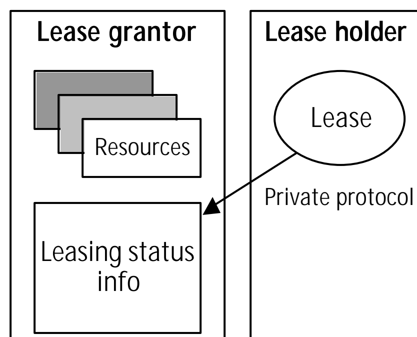

Figure 4-10. Lease interface is used for communication between the grantor and the holder [Edw99].

The management of several leases is very common among resource consumers. This happens especially with lookup service registrations and event notifications. As leasing is a heavyweight programming model, programmers need tools for efficient lease management design. One way to reduce communication needs, computation time and the amount of management code is to use the `LeaseMap` interface for *batching* leases together [Edw99]. With this interface, the individual leases are grouped together and all the actions are directed at all leases in this data structure called `LeaseMap`, which is inherited from the `java.util.Map` class. However, only leases granted by the same grantor can be batched together (Example 4-6) because grantors use private protocols for communication with lease holders as seen in Figure 4-10.

```
long duration1 = duration_for_lease1;
long duration2 = duration_for_lease2;
lmap = lease1.createLeaseMap(duration1);
if (lease2.canBatch(lease1)) {
  lmap.put(lease2, (Long) duration2);
} else {
  // leases can not be batched together
}
```

Example 4-6. Testing a lease for batching.

4.4 Remote Events

Remote events are also a fundamental feature of the Jini architecture. Jini uses events in much the same way as legacy *Java Foundation Classes* (JFC) and AWT based Java applications, but the main characteristics of events is in remoteness between different VMs and machines. In this chapter these differences are reviewed and a basic description of remote events and their usage is given.

4.4.1 The Nature of Remote Events

An event is an abstract change in the internal state of an object. According to the principles of OO programming, this kind of change is encapsulated in an object and is not allowed to be visible outside the object [KoK96]. At any rate, in

Jini system, other objects can *register* interest in such events, and the object can *notify* interested objects when a particular event occurs. This notification, which is in practice an instance of the `RemoteEvent` class, holds enough information about the event to enable further actions of the notified component (Figure 4-11).

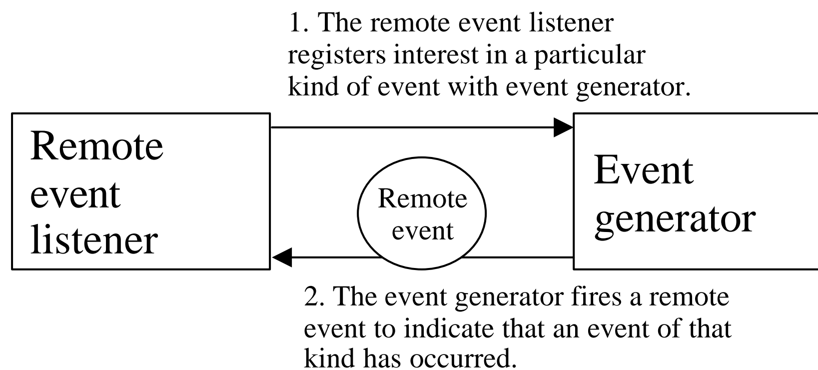

Figure 4-11. Jini's basic event model [Arn99b, Sun99c].

The basis of the Jini event system is that all events are instances of the same standard `RemoteEvent` class or its subclass. Secondly, the Jini APIs for event delivery are very narrow, which makes programming relatively easy.

The notification system used in Jini is called *asynchronous notification*. This means that event notifications are sent directly to the listening software component and the handling of notifications is done outside the control flow of the component by *listeners* [Edw99]. The benefit of this kind of notification handling is that continuous *polling* for events is not necessary and the component is automatically informed when an interesting event occurs without any extra programming code needed [Fur99]. Another advantage is that when the events are handled by some other component, the design of the interested component is more clearer [Edw99].

In Jini the events are classified into *kinds* [Arn99b, Sun99c]. This means that there are one or more kinds of events, which can be fired by a component. The classification of events is implementation dependent. The registration of interest indicates the kind of event that the interested component is interested in. Event types are uniquely identified with the combination of an event identifier and an

object reference. The basic content of the event class (Example 4-7) is the following [Sun99c]:

- A reference to the object in which the event occurred.
- An identifier for the kind of event in which an interest has been registered.
- A sequence number identifying the instance of the event type.
- An object that was passed in, as part of the registration of interest in the event by the registrant.

```
public class RemoteEvent extends java.util.EventObject {
  public RemoteEvent( /* constructor */
    Object source, /* a reference to the object */
    long eventID, /* an identifier of the kind of event */
    long seqNum, /* a sequence number of the event type */
    java.rmi.MarshalledObject handback) /* an object, which
      is handed back when the notification occurs */
}
```

Example 4-7. The constructor of event class.

Using the remote event model is one way to enable implementation of Jini's spontaneous networking. The lookup service is one natural object for using events to notify registered clients that new services are available in the community. An example of such usage could be a digital camera, which wants to print a picture, and if the printer is not online, the camera can register interest for printing and the lookup service can notify the digital camera when the printer is available.

Although polling for events is not necessary, the interest in particular kinds of events must be leased [Sun99c]. The object that wants to receive event notifications must show continuous interest for them, because the whole system needs to maintain its reliability and self-healing abilities. If the object for some reason goes temporarily offline, it can still get the notifications it needs, if they are handled by third-party objects, which are discussed in the next section.

4.4.2 Design Issues

In legacy Java applications, events are usually GUI based and local. These kinds of events are very reliable, easy to program and their delivery is fast and efficient. Efficient delivery means that the time needed to actually handle the event, is relatively much longer than the time needed to send the event. This is true, because sending involves nothing but the invocation of the local event handling method. In distributed systems the situation is reversed [Edw99]. This is why the notification traffic in the network should be well designed and minimized.

There are some other things that must be taken into account when designing a distributed Jini system. In a local system, the programming of ordering event notifications is natural. But in a distributed system there is no such guarantee that notifications arrive in the intended order, because of partial failures and network latency. Network failures are another reason for why there is no general guarantee for notifications to arrive at all. If the local application, inside one VM, sends an event notification, the delivery is sure unless there is a catastrophic failure in the entire application.

There is no global namespace for event identifiers [Edw99]. Hence, all event generators should have their unique way to generate them. When considering, for example, a lookup service, this is a great help when customizing event handling related to client registrations. Sequence numbers are another issue to be considered. To ensure that event notifications are *idempotent*, which means that components receiving notifications are able to deal with them only once regardless of how many times they get the same notifications [Edw99], the sequence numbers of events, generated by the event generator, must meet the following requirement [Arn99b, Sun99c]:

$$x_{im} \textit{ occurs before } y_{in}, \textit{ iff } m < n$$

where

$$x_{im} = \textit{event } x \textit{ with identifier number } i \textit{ and sequence number } m$$

$$y_{in} = event\ \ y\ with\ identifier\ \ number\ i\ \ and\ \ sequence\ \ number\ n$$

Further, if the notification calls need to be guaranteed to be *fully ordered* [Arn99b, Sun99c], meaning that no sequence numbers are skipped, there must be exactly in force that there is

$$n - m - 1\ events\ between\ events\ x_{im}\ \ and\ \ y_{in}\ \ when\ m < n$$

In Jini, it is not specified how services can advertise their event sequencing guarantees [Edw99]. It can be dealt with by adding an extra attribute to the proxy object of the service, when it is registering with the lookup service. Idempotency and increasing sequence numbers within the same kind of event type are still always required.

When designing an event listener class, the following points must be taken into account [Edw99]:

- The listener must be inherited from `UnicastRemoteObject` class, because otherwise its methods cannot be called from other JVMs via RMI. This ability is mandatory to deliver events within TCP streams [Lan99, Sun99f].
- The listener must implement the `RemoteEventListener` interface to support the `notify()` method. It must be noticed that this `notify()` method is not the same thing as the `java.lang.Object.notify()` method, which wakes up a waiting thread [Sun99f].
- The constructor of the listener class and all its remotely callable methods must be able to throw `RemoteException` to work correctly with RMI [Sun99a].

It is very important that all these requirements are met, because the Jini event architecture is the only model, along with transaction participating [Sun99c], in Jini that really requires the use of RMI. Other communications like leasing and registering for lookup service do use RMI, but they can be implemented by using other communication substrates if necessary. However, all exceptions

must be mapped into `RemoteException` instances. The use of other communication systems is possible, because these components communicate via proxy objects, unlike components in the Jini event model.

When designing the `notify()` method of the listener, it should be emphasized that RMI itself uses *synchronous calls* [Edw99], although the whole notification system is asynchronous. This is why the implementation should be able to execute very fast at runtime, because the code that made the remote invocation is blocked during the execution of the remote call. This naturally prevents the service from sending new notifications to other listeners. One solution could be queuing received events for a later processing and returning back to the event generator code as quickly as possible.

There is no standard way or interface in Jini to implement how to register for interesting events, since there is no single way to identify all possible events and their representation inside a particular component [Arn99b]. However, some guidelines can be used when designing such a mechanism. Firstly, when the registration is confirmed by the event generator object, the lease for interested events can be returned at the same time to reduce network traffic. Secondly, the `MarshalledObject`, which is handed back to the event registrant, should initially be passed to the event generator when registering for some event, because it can contain additional information about the event [Arn99b] and can be used to build client-specific events [Edw99].

In a Jini system, services may be inactive for long periods of time. This has to be kept in mind, when sending event notifications to such objects. The sending *policy* must be defined depending on the nature of the service and entire system. Possible policies may include:

- Keep on sending event notification, until it is delivered or until tried a predefined number of times, after which the notification is discarded or stored.
- Discard the event notification that cannot be delivered.
- Store the undelivered event notification and send it again later.

A listener can be added to the Java software component in a number of ways. To make the design of the interested component more maintainable and readable, it is possible to use inner listener classes [Hun99] or adapter classes instead of straightforward listener interface implementation [Cam98]. The benefit of these approaches is a little smaller in Jini, because all the events are instances of the same class and the listener interface implementation only has to implement one method. In the Jini event system the registrant (interested component) and the event listener can be separated (Figure 4-12).

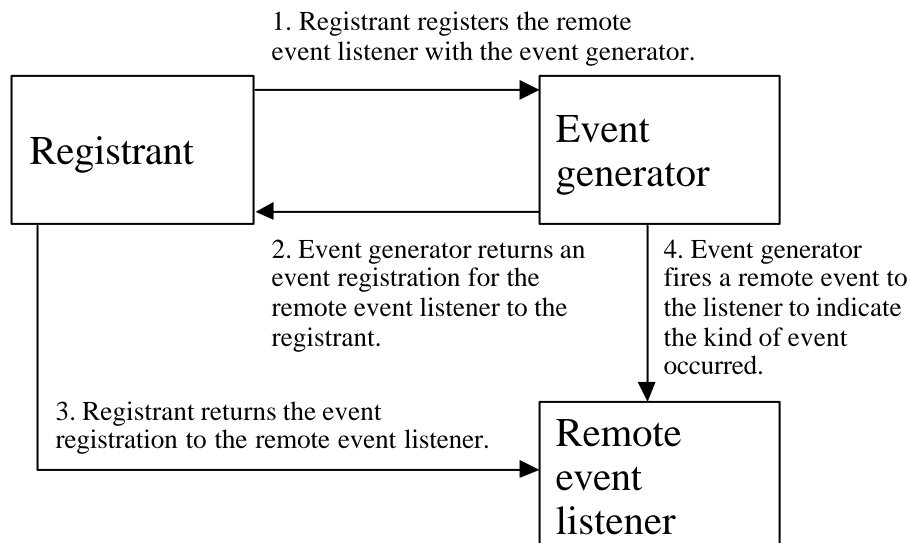

Figure 4-12. Separating the listener and the registrant object [Arn99b].

This is why Jini's event architecture is very simple. Jini supports the ability to add *generic third-party event listeners* (sometimes called agents or event delegates) and chain them together (Figure 4-13). In this connection, the term generic means that third-party listeners can function with events they have not earlier been in touch with. Because all event notifications are instances of the same `RemoteEvent` class, the required QoS properties [Cou94, Far98] can be added to the event system with third-party listeners, which can be used, for example, for enabling delivery guarantees, notification storing, filtering and rerouting purposes [Sun99c]. Third-party listeners do not have to know what kind of events they are really handling and where they come from, contrary to legacy Java systems, where every kind of event has its own type of listener (Figure 4-14). Even if there is some loss of expressiveness [Edw99], the narrow interfaces are needed because of these third-party listeners. On the other hand,

because Java is type based programming language, it supports also other RMI based solutions for distributed events, like the *Event Notifier* model [Gup98, Lan99], which is a proxy based *design pattern*.

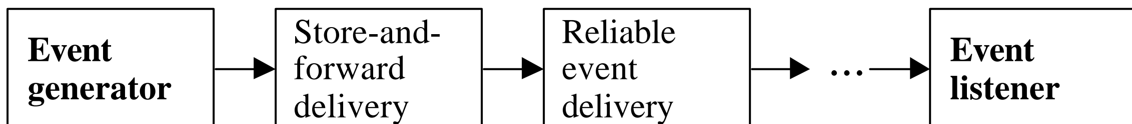

Figure 4-13. Composing event delegates.

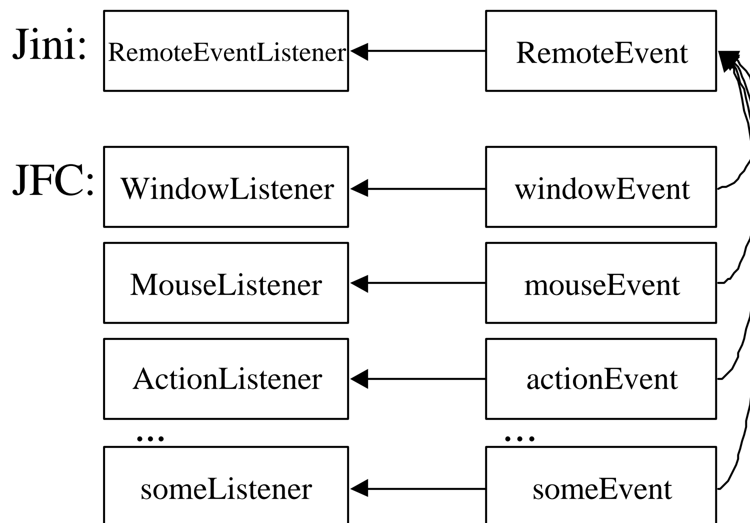

Figure 4-14. Jini forces all remote events to be instances of the `RemoteEvent` class.

When designing a third-party listener, the first thing is, that it must implement the `RemoteEventListener` interface to support the `notify()` method [Edw99]. Because these listeners act like services, they must be able to process event notifications from multiple sources and deliver them to multiple receivers. To meet this requirement, the third-party listener must know beforehand, what kinds of events and what source a particular receiver wants event notifications from. This information is delivered within a registration process by the receiver and the mapping can be accomplished (Figure 4-15).

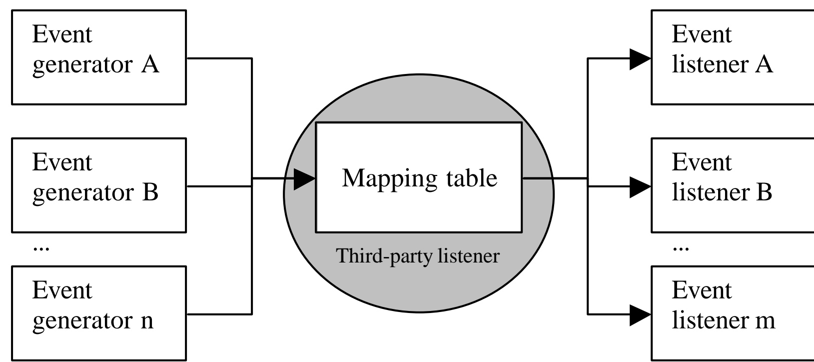

Figure 4-15. Event routing from multiple generators [Edw99].

The route through which an event notification is transmitted from the generator to the final receiver is called a *pipeline*. Creating the pipeline is easier from the generator's point of view, because it usually registers only with one event delegate [Edw99]. When the receiver registers with the generator, the listener is passed to it. This is the mechanism which is also used when the generator registers with the particular event delegate by calling the `register()` method (Example 4-8) and so the pipeline is created.

```
class SomeDelegate implements RemoteEventListener {
  //...
  void register(Object source, /* event generator */
    long eventID, /* event type identification */
    RemoteEventListener downstream); /* event receiver */
  //...
}
```

Example 4-8. `register()` method parameters of an event delegate.

The situation is different, when the event receiver has to create the event pipeline for some reason. This is more complex, because the receiver needs to register with the most upstream event generator and work downstream [Edw99]. The difficulty is in providing each event delegate with appropriate event routing information for each stage. Another problem is that the generator may send event notifications before the pipeline has been completely constructed. In these situations, the policy for sending events with unknown destination must be defined within the event delegate.

As mentioned before, all receivers wanting to get event notifications, must lease this ability. Leasing can be done at least in two different ways (Figure 4-16). In the parallel model the receiver leases all delegates and the generator when it is registering with them. With this kind of strategy the leases are managed well with the `LeaseRenewalService` class [Sun99b] or some other suitable tool. The serial model is closer to the Jini-style generator-listener approach, where the listener is responsible for renewing leases for registered services offered by the generator [Edw99]. One of the advantages is that the receiver can be inactive for long periods of time and still get event notifications relatively easy. The drawback of this model is that when the receiver crashes, the time needed to cancel the leases and detect the failure in the pipeline will be longer than in the parallel model.

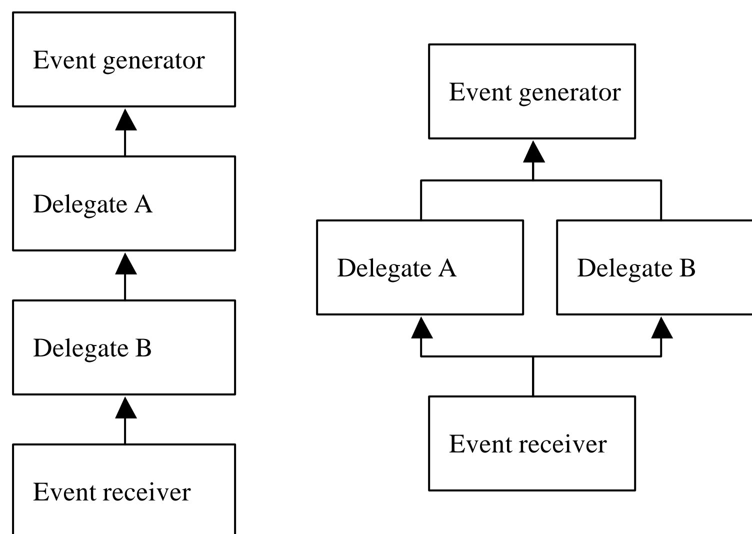

Figure 4-16. Lease maintaining possibilities (serial and parallel) with event delegates from the event receiver's point of view.

The use of third-party listeners has many practical advantages. An event generator, which uses some third-party listener, can be designed in a clearer way. Because all the code needed to deliver events, for example, in a strict and particular order, can be placed within the listener instead of the generator. This is also helpful, when reusing program code in the OO way, because third-party listeners can be considered as services and can be used as *plug-ins*. Through listeners the clients can have extra services, which cannot be offered by the

generators. One implementation of this kind of QoS feature is event notification multiplexing (Figure 4-17).

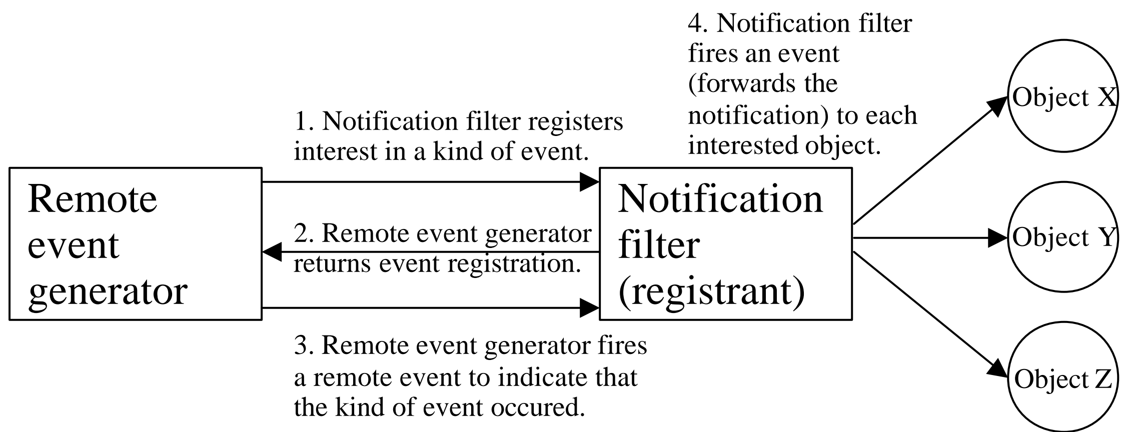

Figure 4-17. Event notification multiplexing [Arn99b].

Filtering is also useful, when the whole event system is being simplified in the other direction. A filter can collect particular events sent by objects, store them and notify the interested component, when all predefined events have occurred (Figure 4-18). Especially when a component is registering with the lookup service [Edw99] to request some general service, like persistent storage, and it is not possible to set exact attributes for service template matching, filtering can be used to sift services, which do not fulfill the conditions set by the component.

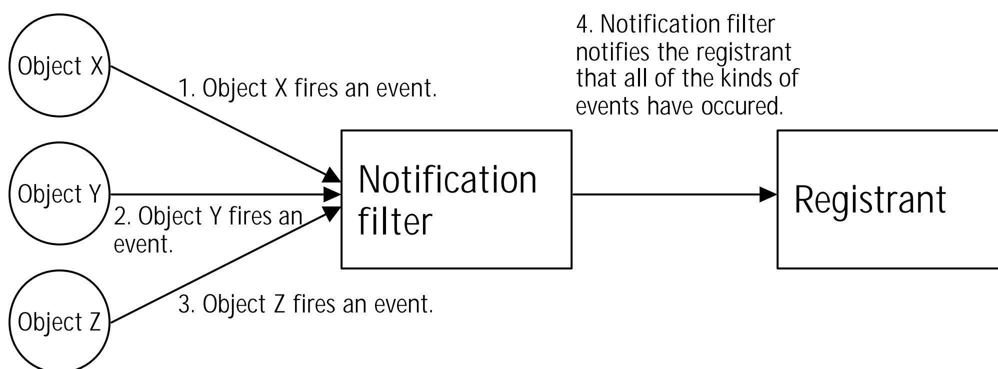

Figure 4-18. Event notification demultiplexing [Sun99c].

One promising feature in the future would also be the integration between remote events of Jini and EJB technology. This has been considered already in [Sun99c]. Even if EJB technology is mostly focused on the server side and the

Jini model more on the client side. They make a powerful combination [Kle00], because both of them support the concept of distributed component interaction [Swa99] in a platform and application independent way [Won98]. On the other hand, some of the differences are compared in Table 4-6.

Compared subject	**Jini**	**Enterprise JavaBeans**
Architectural model	Peer-to-peer	3-tier
Typical platforms	Java client / device	Java application server / Java legacy server
Typical topology	Distributed client	Distributed enterprise
Protocols	RMI / Discovery / Join	*Internet Inter-Orb Protocol* (IIOP) / *Java Messaging Service* (JMS) / CORBA

Table 4-6. Comparison between Jini and EJB [Kle00].

Implementation for mapping Jini events to EJB events requires at least a conversion from EJB event object to `RemoteEvent` object supported by Jini, and vice versa [Sun99c]. Because Jini uses the notification paradigm, the events passed to the EJB event listeners should also be converted as appropriate method calls.

4.5 Transactions

Jini transactions are used when multiple components need to perform a task together in a cooperative way. This transaction model is the least used model in Jini [Edw99], because its use is optional even in JavaSpaces. Distributed transactions are still one of the ways to deal with partial failures. This model also aims to maintain the known state of the communication among the participants during collaborative actions. All actions inside one transaction must either succeed or fail [Arn99b, Sun99c].

In this section, the participants of a transaction are looked at and some differences between the traditional transaction protocol and the Jini transaction protocol are discussed.

4.5.1 The Nature of Transactions

In traditional transaction based systems, like database systems, the *two-phase commit protocol* (2PC protocol) is widely used [Orf96]. The Jini transaction model is based on the same protocol. In fact, this protocol and certain interfaces are the only things that have been standardized in Jini. Everything else is system specified and must be implemented according to the functionality requirements of the services.

Transactions must fulfill the *atomicity*, *consistency*, *isolation* and *durability* (ACID) properties (Table 4-7) [Cou94, Edw99]. However, these requirements are quite strict to be fulfilled every time when using the 2PC protocol. This is why the implementation of these properties is left to the individual components in Jini [Sun99c].

Atomicity	All or nothing principle [Cou94]. All the actions within the transaction must complete. If this does not happen, no action is committed [Sun99c].
Consistency	After the transaction, the system must be in a consistent state. This state is abstract and only known by humans (for example; an employee always has a manager). The transaction itself does not guarantee this, but it is a tool to achieve it [Sun99c].
Isolation	Ongoing transactions are not allowed to have an impact on each other, which means that participants are not allowed to examine or use intermediate results [Arn99b, Sun99c].
Durability	The results of transaction should be at least as persistent as the object that commits the transaction [Arn99b, Edw99].

Table 4-7. ACID properties.

Distributed transactions differ from single system transactions in the same way as every other distributed computing paradigms. The greatest problem is that in distributed systems some component has a possibility to interact with multiple other components, which can deal with the same transaction in different states [Sun99c].

In Jini, a transaction has three conceptual copartners (Figure 4-19). Like in all 2PC protocol based systems, there is a manager, which controls transactions and steps through the stages of the protocol. The manager is actually a Jini service, which can handle multiple transactions simultaneously and can be shared among members of the Jini community [Edw99]. The manager keeps a list of all participants. The management service must be leased by the clients [Sun99c]. The manager uses the `TransactionFactory` class to create new transactions, which are returned from the factory to the client with the transaction object including the interface, ID number and the lease [Sun99c].

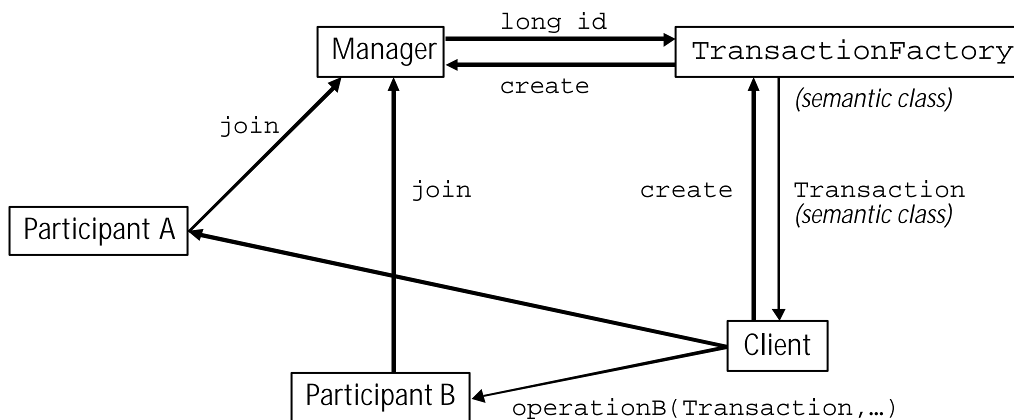

Figure 4-19. Transaction participants [Arn99b].

Clients are components, which want to perform some grouped actions that seem to happen at the same time, and in the limited address space. The client is the component that starts a new transaction by calling the transaction manager service, which it has discovered from the lookup service. After the client has downloaded the object, which implements the `Transaction` interface, it can ask the manager to join wanted participants to the transaction through the interface just provided by passing object references to the necessary services using the `join()` method. The participant is then returned an object, which has the `TransactionParticipant` interface, which is used for the communication with the manager. The interface is provided to the participant by the manager, when the client asks the participant to perform desired actions for the first time [Arn99b].

4.5.2 Design Issues

One of the features of the 2PC protocol is to separate the transaction semantics from the completing protocol used for implementation. In practice, the semantics is represented as return values and parameters of the interacting methods.

As seen before, the purpose of the transaction is to maintain the known state of the communication and try to deal with partial network failures. The states and progression of transaction are presented in Figure 4-20 and Figure 4-21 from all three points of view.

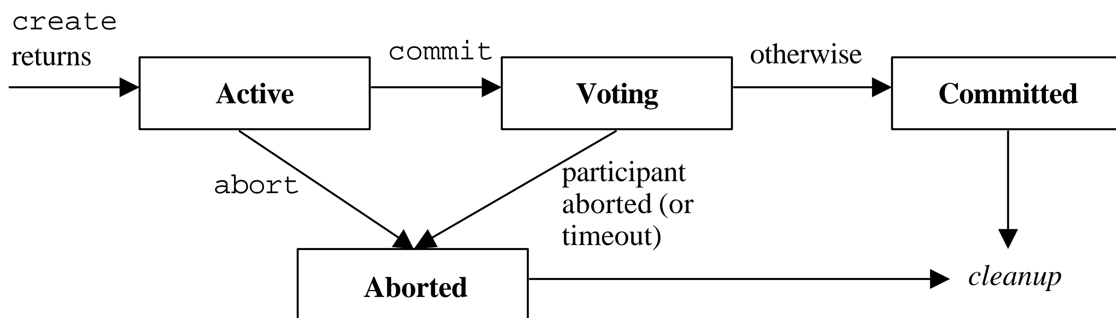

Figure 4-20. Completing the transaction; client (manager) [Sun99c].

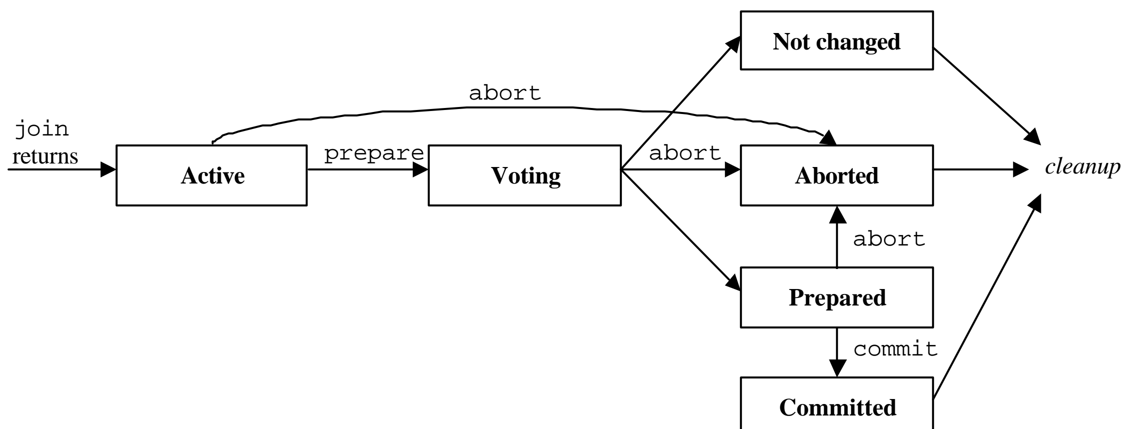

Figure 4-21. Completing the transaction; participant [Sun99c].

For the client the transaction is *active* after the `create()` method returns. The client can either *commit* or *abort* the transaction by notifying the manager about the decision by using the corresponding methods of the proxy object of the

manager [Arn99b]. After the commit decision, the manager performs *voting*, which forces participants to continue to the next phase of the transaction, if all the participants can perform *prepare*. Otherwise the transaction is aborted.

From the manager point of view, the situation is the same as just after the creation of transaction. When the client invokes the `commit()` method, the manager asks all participants to prepare to vote by invoking `prepare()`. Possible return values for voting are *notchanged*, *aborted* or *committed*. If any of the participants vote aborted, the manager must abort the transaction. In this point it must be noticed that if the return value of `prepare()` is `RemoteException`, the manager must keep on retrying according to the application specific policy, because the state of the transaction within the participant is not known. In the situations, where the return value is either `UnknownTransactionException` or `NoSuchObjectException`, which both means that the participant has lost the state of the transaction [Sun99c], and the abortion should be immediate. The manager performs the abortion by announcing the participants. This is a better way to abort instead of letting the leases of participants to expire, because the resources are freed faster.

The transaction manager will move the transaction to a committed state in two ways: 1) When none of the participants vote for abortion, and at least one is in the prepared state (Example 4-9), or 2) all participants vote notchanged, which means that no action after the last voting has been taken, and the whole transaction can be committed. The committed state of the transaction means that all actions have been taken and all decisions have been *rolled forward* [Sun99c].

```
public int prepareAndCommit(TransactionManager mgr, long
  id) throws UnknownTransactionException, RemoteException
{
  int result = prepare(mgr, id);
  if (result == PREPARED) {
    commit(mgr, id); // roll forward all changes
    result = COMMITTED;
  }
```

```
  return result;
}
```

Example 4-9. Basic semantics of the participant's `prepareAndCommit()` method [Arn99b].

Certain states within the transaction are called *commit points*. These are states that must be saved in some fashion into a persistent storage, when they occur. This is done because of the *crash recovery* ability, which means that the particular state can be recovered after a system crash or a network failure, if the transaction is not committed yet. The manager should save the state when it enters to a committed state with at least one prepared participant [Sun99c]. The participant stores the state of itself when it votes prepared, so it can perform roll forward in the time of future invocation of the `commit()` method, even if the system crashes in between. After the successful invocation of the `commit()` method, this storage can be removed [Arn99b]. These commitments require the implementation of the manager and participants to be done using some durable forms of RMI references [Sun99c], like class `Activatable`, which offers remote object support for over time persistence and system activation [Cam98].

When designing transactional functions in Jini, there is a possibility to *nest* transactions [Arn99b]. Nested transactions are *subtransactions* that can exist inside top-level transaction to help accomplishing more complex tasks. The semantics resembles normal transactions, but the nested transaction takes part in the *parent* transaction. The manager of both kinds of transactions can be the same entity.

Another feature of transactions are *deadlocks*. A deadlock means a situation, where none of the cooperating entities can proceed, unless some of the interacting entities free the needed resource first, and they cannot do that, because they again need some other resource, which is also not free. The Jini transaction model cannot guarantee to prevent or even detect deadlocks [Sun99c]. If a deadlock occurs, it is permitted that the participant and the manager can break them by aborting the transaction. This situation leads to the state, where all the interacting entities perform *roll back*. This means that all performed actions are cancelled, and the entities and the whole system within

the transaction participants should be able to return to the state, which existed before the aborted transaction.

5. DISCUSSION

To sum up, Jini provides an extensive framework for developing flexible and robust distributed systems. It seems to fulfill the expectations related to network administration work reduction in a time, when the complexity of the distributed systems keeps growing. It also deals with network failures better than the traditional object-based distribution solutions, because of its powerful and careful interface design.

A number of different distributed system frameworks exist as introduced in [Mea00], [Rek99] and [Sha99] in addition to the ones that where shortly mentioned in this thesis. The interoperability between these techniques is a key question, when considering the global development work in the future. It seems that Jini is quite adaptable to these other solutions because of its flexible nature, but there is a lot of competition going on between vendors like, for instance, Sun Microsystems and Microsoft Corporation. The worst scenario is that the techniques of these competing vendors try to drop each other out from the market instead of trying to complement each other. In this kind of competition, the best technology cannot always survive. More discussion about this rigid competition can be found in [CID99b]. Some research is already ongoing related to the complementing issues, and the results seem to be very promising [Gut99b], especially with Jini and SLP [Mat99].

In addition to the solutions for the challenges of distributed computing, Jini offers capabilities, which are required to fulfil the needs of modern end users of the network services. Flexibility and spontaneity are the most important issues, which are possible to deploy, if a Jini based system is designed in a sophisticated way. As the degree of distribution of the services grows all the time, security issues are also very important. In this area, the developers of Jini still have a lot of work to do, because Jini does not provide any security mechanisms in addition to the basic Java environment properties.

5.1 Drawbacks

At any rate, some drawbacks for Jini exist. One of the most restricting ones is that Jini is not compatible with the *Kilobyte Virtual Machine* (KVM) [Sun99j],

which is a very limited VM. It is targeted to embedded devices with approximately 128 kilobytes of available memory. There exists a lack of interoperability, because Jini requires some J2SE based features like RMI to work, and these kinds of features are too heavyweight for the current KVM implementations. The role of the KVM is to be a part of the new running environment of the larger developing solution J2ME for limited-resource mobile devices. This development environment is constructed of *configurations* and *profiles* [Day00b] (Figure 5-1). Configurations include a VM, optimized APIs and core classes for a certain type of a device, such as a PDA or a mobile phone, which is not capable of running a full-blown JVM. Current basic configuration for the KVM is called *Connected, Limited Device Configuration* (CLDC) [Sun00a], which has been already approved by the *Java Developer Community* (JDC) as a *Java Specification Request* (JSR).

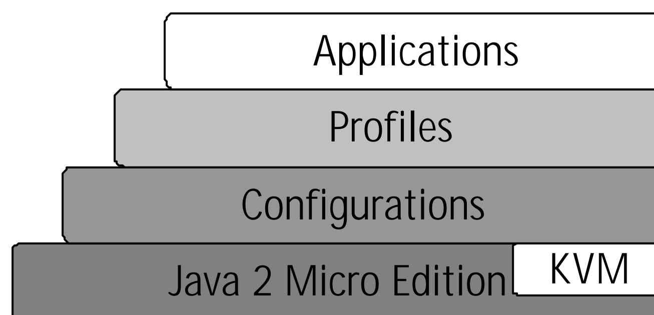

Figure 5-1. Configurations and profiles atop J2ME.

Profiles extend configurations by adding APIs that provide more capabilities for a specific market segment and a device type. The most interesting profile at the moment is the *Mobile Information Device Profile* (MIDP) [Sun00b], which is targeted to mobile, wireless, bandwidth-limited, battery operated devices, as for instance, mobile phones. Jini support for these kinds of platforms, like J2ME, is still not available. This has reduced the success and adoption of the Jini technology, because wireless devices are the ones, which are now being integrated as a part of existing, IP based distributed systems.

As a solution for capacity problems, there exists a new *surrogate* technology [Sun00c]. The principle of this technology is that some other Jini network-connected device, which has better capabilities to execute code, acts on behalf

of the limited devices, which are connected to these *surrogate hosts* through interoperable *adapters* using a *surrogate protocol*. The model has much equivalence with the third-party remote event model and the leasing model used in Jini. In addition, the surrogate technology seems to be the only solution at this moment for *Bluetooth* (BT) [Blu99] and *Wireless LAN* (WLAN) [IEE99] enabled devices, which will soon exist on the mass market. It seems natural that BT and WLAN *access points* (APs) will work as surrogate hosts, since they can be static and have more processor power to run the full JRE in support of the wireless devices itself. The combination of these wireless technologies and Jini is definitely a worth of investigating in the future.

The hardware and software requirements discussed in the thesis can already be met to some extend within the limited devices, but the market seems not to be ready for Jini products yet, at least not the mass market of cost-critical consumer products. Nevertheless, some attempts to break into the market have already been made, for instance [Nie00], and the development of the new concepts is ongoing in the laboratories of various companies. One general example for embedded devices is reviewed in [McD00].

In addition to the relatively high computing capacity requirements, another drawback seems to be the security issues of Jini systems. The highly mobile nature of objects requires new security paradigms to be used to protect the data and the privacy of end users. As described by [Cri99], traditional security models are class-based, but they should be object-based, because the objects are the ones that are transferred over the network. Even if the JVM with security policies offers the built-in security for Java objects, it does not seem to be enough for the Jini environment, because Jini expands Java's distribution and code downloading capabilities, but offers no extra security. Moreover, in Jini there is no default authentication mechanism in the lookup service specification, and furthermore, the discovery process is not protected, even if the use of services is controlled. Some solutions to security hazards in Jini systems are presented in [And00], and [Cze99] presents a Jini capable architecture, which is based on a secure *Service Discovery Service* (SDS) with the use of *eXtensible Markup Language* (XML) [W3C97].

5.2 Future

In general, Jini specifications are relatively loose, which is an advantage, because it gives developers room for being creative and innovative. The specifications define mostly behavior related issues, which has to be taken care of when implementing the services. All well-behaving Jini services should be capable of fulfilling the requirements defined by the specifications. These requirements have been discussed along with the discovery and join protocol issues.

It can also be a disadvantage to have so many system specific issues in an implementation process. Without well-guided general development paths, Jini based services can vary a lot in the future. Of course this is also desirable from the content point of view, but the difficulties are on the service interface side. In theory, an infinite number of possibly different services must be able to be implemented, but still all of them should be reachable by all potential clients through their known interfaces. Standardization process for these interfaces is an enormous task, which requires very careful designing.

Jini's integration with the intelligent agent software technology is an interesting development path. As the Java programming language has become common, the portability problem of agent components has been solved for some extent. Also a VM based code execution provides better security solutions against hostile agents. Firstly, some guidelines are needed to model a Jini service as a software agent to enable even more intelligent and adaptable solutions. Secondly, Jini does not really define a standard way for clients and services to communicate. However, to be interoperable with general software agents, the interaction of components must be clearer as it is now. One solution is to integrate Jini services to be able to communicate with agents through *Agent Communication Language* (ACL). Some ways to make this happen are presented in [Ash00] and [Che99]. Because many agents are *autonomous*, which means that they do not need any human intervention to work, the Jini technology provides a good development framework for the mobile agent technologies because of spontaneity and a lightweight service discovery. The

advantages mentioned here are obvious, when compared to the former work on this area, for instance, presented in [Cha98].

We have a lot of electronic devices around to help us out scheduling meetings, locating restaurants and sending messages etc. These devices are usually focused on carrying out a particular task e.g. playing music or warming up the food. The realization of the concept of ubiquitous computing is only partly accomplished, because the technology is not invisible to the end users yet. The trend of the future seems to be that the end users are not required to command the devices to do some task, but the devices perform the task independently according to the current needs of the users. This is a goal, because the large number of alternative services is impossible to handle manually, even at this moment. Some very interesting visions of this kind of technology behavior are presented in [Esl99].

Agent technologies have a significant role in the development of the ubiquitous computing. If considering Jini as a framework for agents, it is noticed that, in all probability, this way Jini has a role in ubiquitous computing too. As predicted in [Esl99] and [Kon00], data and services will be more important than location, which means that the end users must be able to connect to the network for the service in spite of the location of it and users. This, again, fits nicely to the Jini's idea of a service-centric network. This location independent approach has in recent years evolved fast, since the popularity and service development of mobile phones have increased. On the other hand, local improvements are still far away from the target of ubiquitous computing vision, because only the first steps has already been taken with BT and WLAN.

It is clear at the moment that Jini is only suitable for a LAN environment, which has to be adaptable enough to handle clients, which temporarily connect to the LAN for some service and then leave the network. Furthermore, a Jini system is not scalable in the full meaning of the term, because it can only grow from a LAN to a WAN by using the ability to federate the lookup services. This will definitely chance in the future after the new versions of Jini are introduced as discussed along this thesis. At any rate, one problem related to WANs is that

the service discovery by using the multicast messaging model is obviously not an option.

Despite of the drawbacks, Jini is evolving all the time. Very important thing for Jini technology has been Sun Microsystems' new model for developing technical concepts further [Wal00]. Jini is so called *open source technology*, which means that anybody can participate to the development process by e.g. reviewing the specifications, proving comments and suggestions. Sun Microsystems has succeeded well with this model, and it seem that they have been able to create a very active community of developers with more than 40000 members [Mor00]. All service developers use the *Sun Community Source License* (SCSL) model, which is a way to provide fast access to the technology and retain developers' intellectual properties related to Jini code, but it still provides some income to the original developers of Jini, when community members make any commercial products.

Even if Jini still does not fulfill all the wild ideas of the ability to connect every kind of devices, like toasters, refrigerators and wristwatches, to each other and to the Internet, it certainly offers more down-to-earth advantages to the distributed computing as seen. The key features of Jini, e.g. simplicity and spontaneity, will be even more important in the future as the number of various systems, networks, clients, products and services are being developed.

6. REFERENCES

[Abd98] Abdullahi, S. E. and Ringwood, G. A.: Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Computing Surveys*, 30(3): 331-373, 1998.

[And00] Andersson, F. and Karlsson, M.: *Secure Jini Services in Ad Hoc Networks*. Master of Science Thesis, Royal Institute of Technology, Stockholm, Sweden, 2000.

[Arn98] Arnold, K. and Gosling, J.: *The Java Programming Language, Second Edition*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1998.

[Arn99a] Arnold, K.: *The Jini Architecture: Dynamic Services in a Flexible Network*. Proceedings of the 36th ACM/IEEE Design Automation Conference, IEEE Computer Society, 1999, 157-162.

[Arn99b] Arnold, K. & al.: *The Jini Specification*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.

[Ash00] Ashri, R. and Luck, M.: *Paradigma: Agent Implementation through Jini*. Proceedings of the 11th International Workshop on Database and Expert Systems Applications, 2000, IEEE Computer Society, 2000, 453-457.

[Aye99] Ayers, D. & al.: *Professional Java Server Programming*. Wrox Press Ltd., Birmingham, United Kingdom, 1999.

[Bec99] Beck, J., Gefflaut, A. and Islam, N.: *MOCA: A Service Framework for Mobile Computing Devices*. ACM International Workshop on Data Engineering for Wireless and Mobile Access, ACM, 1999, 62-68.

[Blu99] Bluetooth SIG: *Specification of the Bluetooth System Version 1.0 A*. Bluetooth SIG, 1999.

[Bry99] Brydon, S. P. & al.: *Javelin++: Scalability Issues in Global Computing*. ACM 1999 Conference on Java Grande, ACM, 1999, 171-180.

[Bus98] Buss, A. and Jackson, L.: *Distributed Simulation Modeling: A Comparison of HLA, CORBA, and RMI*. 1998 Winter Simulation Conference (ed. Medeiros, D. J. & al.), ACM, 1998, 819-825.

[Cam98] Campione, M. and Walrath, K.: *The Java Tutorial Second Edition*. Addison-Wesley Publishing Company, Inc., Padstow, Great Britain, 1998. Available at: http://java.sun.com/docs/books/tutorial/ (28th of February 2000).

[CaD98] Caromel, D. and Vayssiere, J.: *A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming*. ACM 1998 Workshop on Java for High-Performance Network Computing, ACM, 1998.

[CaB99] Carpenter, B. & al.: *Object Serialization for Marshalling Data in a Java Interface to MPI*. ACM 1999 Conference on Java Grande, ACM, 1999, 33-38.

[Cha98] Chauhan, D. and Baker, A. D.: *JAFMAS: A Multiagent Application Development System*. Second International Conference on Autonomous Agents, ACM, 1998, 100-107.

[Che99] Chen, H. L.: *Developing Agent Oriented Jini Services*. The Second Jini Community Meeting, The Jini Community, 1999. Available at: http://www.jini.org/comm2/jcm2_agents+jini_chenpaper.pdf (27th of November 2000).

[ClD99a] Clark, D.: Network nirvana and the intelligent device. *IEEE Concurrency*, 7(2): 16-19, 1999.

[ClD99b] Clark, D.: Service with a (smart) smile: network Jini-style. *IEEE Intelligent Systems*, 14(3): 81-83, 1999.

[ClE99] Clark, E. (ed.): Mobile Code Safety. *Network Magazine*, 14(12): 37-53, 1999.

[Com00a] Comer, D. E.: *Internetworking with TCP/IP*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2000.

[Com00b] Comer, D. E. and Stevens, D. L.: *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2000.

[Cou94] Coulouris, G., Dollimore, J. and Kindberg, T.: *Distributed Systems*. Addison-Wesley Publishing Company, Inc., Padstow, Great Britain, 1994.

[Cri99] Crichton, C., Davies, J. and Woodcock, J.: *When to trust modile objects: access control in the Jini Software System*. Proceedings of the Technology of Object-Oriented Languages and Systems, 1999 (TOOLS 30) (ed. Firesmith, D. & al.), IEEE Computer Society, 1999, 116-125.

[Cze99] Czerwinski, S. E. & al.: *An Architecture for a Secure Service Discovery Service*. The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1999, ACM, 1999, 24-35.

[Day00a] Day, B.: *Jini Connection Technology Architecture Overview*. Internet WWW-presentation, URL: http://www.billday.com/Work/Jini/jini.pdf (9th of December 2000).

[Day00b] Day, B.: *Java Technology for Mobile Devices*. Developing for Mobile Platforms Session, Wrox Wireless Professional Developer Conference, Amsterdam, The Netherlands, 11th of July 2000.

[Edw99] Edwards, W. K.: *Core Jini*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1999.

[Esl99] Esler, M. & al.: *Next Century Challenges: Data-Centric Networking for Invisible Computing*. The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1999, ACM, 1999, 256-262.

[Far98] Farley, J.: *Java Distributed Computing*. O'Reilly & Associates, Inc., Sebastopol, California, 1998.

[Fre99] Freeman, E., Hupfer, S. and Arnold, K.: *JavaSpaces Principles, Patterns, and Practise*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.

[Fur99] Furr, S.: Real-Time Extensions to the Java Platform. *Embedded Systems Programming*, 12(13): 71-78, 1999.

[Gen99] Genßler, T. and Löwe, W.: *Correct Composition of Distributed Systems*. Proceedings of the Technology of Object-Oriented Languages and Systems, 1999 (TOOLS 31) (ed. Chen, J., Lu, J. and Meyer, B.), IEEE Computer Society, 1999, 296-305.

[Gle00] Gleeson, B. & al.: *A Framework for IP Based Virtual Private Networks*. RFC2764, The Internet Society, 2000.

[Gos96] Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1996.

[Gup98] Gupta, S., Hartkopf, J. and Ramaswamy, S.: Event Notifier, a Pattern for Event Notification. *Java Report*, 3(7): 19-36, 1998.

[Gut99a] Guttman, E. & al.: *Service Location Protocol, Version 2*. RFC 2608, The Internet Society, 1999.

[Gut99b] Guttman, E. and Kempf, J.: *Automatic Discovery of Thin Servers: SLP, Jini and the SLP-Jini Bridge*. Proceedings of the 25th Annual Conference of the IEEE Industrial Electronics Society (IECON '99), IEEE Computer Society, 1999, 722-727 (vol. 2).

[Har97] Harold, E. R.: *Java Network Programming*. O'Reilly & Associates, Inc., Sebastopol, California, 1997.

[Hic99] Hicks, M. & al.: *Transparent Communication for Distributed Objects in Java*. ACM 1999 Conference on Java Grande, 1999, ACM, 1999, 160-170.

[Hol98] Holder, O. and Ben-Shaul, I.: *Dynamic Layout of Distributed Applications*. Third International Workshop on Software Architecture, ACM, 1998, 77-80.

[Hun99] Hunt, J.: Inner Space. *Application Development Advisor*, 2(4): 46-51, 1999.

[IEE99] IEEE: *Standard for Information Technology 802.11*. IEEE Standard for Information Technology 802.11-1999, IEEE, 1999.

[Jah98] Jahan, N.: *Master Thesis: Making Objects Persistent in a CORBA Environment*. University of Helsinki, Helsinki, 1998.

[Jan99] Janssen, B. & al.: *ILU 2.0beta1 Reference Manual*. Internet WWW-page, URL: ftp://ftp.parc.xerox.com/pub/ilu/2.0b1/manual-html/manual_toc.html (25th of August 2000).

[Kap99] Kaplan, L.: Distributed computing in Java: RMI, CORBA, or both? *Java Report*, 4(7): 78-80, 1999.

[Kle00] Kleinman, R.: *Jini and Enterprise JavaBean Technologies--The Distributed Client Meets the Distributed Service*. Internet WWW-page, URL: http://developer.java.sun.com/developer/technicalArticles/jini/ (23rd of February 2000).

[Kon00] Kon, F. & al.: *2K: A Distributed Operating System for Dynamic Heterogeneus Environments*. Proceedings of the Ninth International Symposium on High-Performance Distributed Computing, 2000, IEEE Computer Society, 2000, 201-208.

[KoK96] Koskimies, K.: *Julkaisusarja C: Pieni oliokirja*. University of Tampere, Tampere, 1996.

[KoP99] Kostkova, P. and McCann, J. A.: *MAGNET: An Architecture for Dynamic Resource Allocation*. ACM International Workshop on Data Engineering for Wireless and Mobile Access, ACM, 1999, 77-83.

[Lan99] Landis, S.: Distributed Event Notification Using RMI. *Java Report*, 4(7): 17-28, 1999.

[Lea98] Leach, P. J. and Salz, R.: UUIDs and GUIDs. IETF Internet-Draft. Available at: http://www.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt (7th of August 2000).

[Lew98] Lewandowski, S. M.: Frameworks for Component-Based Client/Server Computing. *ACM Computing Surveys*, 30(1): 3-27, 1998.

[Lin99] Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification, Second Edition*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999. Available at: http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html (17th of November 1999).

[Lon99] Longshaw, A.: Getting Clever with Components. *Application Development Advisor*, 2(4): 12-16, 1999.

[Lun00] Lundin, J.: *Java Technology Evaluation*. Nokia Ventures Organization, 2000.

[Mah00] Mahmoud, Q. H.: *Distributed Programming with Java*. Manning Publications Co., Greenwich, Connecticut, 2000.

[Mat99] Matilainen, P.: *Service discovery with Jini and SLP*. Nokia Research Center, 1999.

[McD00] McDowell, C. E. and Shankari, K.: *Connecting non-Java devices to a Jini network*. Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS33), IEEE Computer Society, 2000, 45-56.

[Mea00] Meadows, J.: An Introduction to the JetSend Protocol. *Embedded Systems Programming*, 13(1): 59-68, 2000.

[Mic97] Microsoft Corporation: *Distributed Component Object Model Protocol – DCOM/1.0 (Internet Draft)*. Internet WWW-page, URL: http://msdn.microsoft.com/library/default.asp?URL=/library/specs/distributedcomponentobjectmodelprotocoldcom10.htm (25th of August 2000).

[Mic00a] Microsoft Corporation: *Universal Plug and Play Device Architecture Version 1.0*. Internet WWW-page, URL: http://www.upnp.org/UPnPDevice_Architecture_1.0.htm (26th of July 2000).

[Mic00b] Microsoft Corporation: *Component Object Model (COM) Specification 0.9*. Internet WWW-page, URL: http://msdn.microsoft.com/library/default.asp?URL=/library/specs/S1CF80.HTM (25th of August 2000).

[Mor00] Morgan, S.: Jini to the rescue. *IEEE Spectrum*, 37(4): 44-49, 2000.

[Mur98] Murhammer, W. K. & al.: *TCP/IP Tutorial and Technical Overview*. International Business Machines Corporation, Research Triangle Park, North Carolina, 1998.

[Nes99] Nester, C., Philippsen, M. and Haumacher, B.: *A More Efficient RMI for Java*. ACM 1999 Conference on Java Grande, ACM, 1999.

[OMG99a] OMG: *Java Language to IDL Mapping*. OMG formal/99-07-59, 1999. Available at: ftp://ftp.omg.org/pub/docs/formal/99-07-59.pdf (19th of November 2000).

[OMG99b] OMG: *The Common Object Request Broker: Architecture and Specification V2.3.1*. OMG formal/99-10-07, Framingham, Massachusetts, 1999. Available at: http://www.omg.org/corba/corbaiiop.html (17th of November 1999).

[Orf96] Orfali, R., Harkey, D. and Edwards, J.: *The Essential Client/Server Survival Guide*. John Wiley & Sons, Inc., New York, New York, 1996.

[Pag97] Page, E. H., Moose, Jr., R. L. and Griffin, S. P.: *Web-Based Simulation in Simjava Using Remote Method Invocation*. 1997 Winter Simulation Conference (ed. Andradóttir, S. & al.), ACM, 1997, 468-474.

[Per99] Perrone, P. J. and Chaganti, V.: Jini in the Box. *Embedded Systems Programming*, 12(12): 55-64, 1999.

[Pis99] Pistoia, M. & al.: *Java 2 Network Security Second Edition*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1999.

[Pos80] Postel, J.: *User Datagram Protocol*. RFC 768, The University of Southern California / Information Sciences Institute, 1980.

[Pos81] Postel, J.: *Transmission Control Protocol*. RFC 793, The University of Southern California / Information Sciences Institute, 1981.

[Rek99] Rekesh, J.: *UPnP, Jini and Salutation – A look at some popular coordination frameworks for future networked devices*. A technology white paper, California Software Laboratories, 1999.

[Sal00] The Salutation Consortium: *Salutation Architecture Specification 2.0c*. Available at: http://www.salutation.org/ (29th of July 2000).

[Sha99] Sharma, R.: *Distributed Application Development with Inferno*. 36th ACM/IEEE Conference on Design Automation Conference, IEEE, 1999, 146-150.

[Slo94] Sloman, M. (ed.): *Network and Distributed Systems Management*. Addison-Wesley Publishing Company, Inc., Wokingham, Great Britain, 1994.

[Sri97] Sridharan, P.: *Advanced Java Networking*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1997.

[Ste98] Steiert, H.-P.: *Towards a Component-based n-Tier C/S-Architecture*. Third International Workshop on Software Architecture, ACM, 1998, 137-140.

[Sun98] Sun Microsystems, Inc.: *PersonalJava Technology White Paper*. Available at: http://java.sun.com/products/personaljava/pj_white.pdf (31st of July 2000).

[Sun99a] Sun Microsystems, Inc.: *Java Remote Method Invocation Specification*. Internet WWW-page, URL: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html (8th of March 2000).

[Sun99b] *Jini Technology Helper Utilities And Services Specification 1.1 Alpha*. Sun Microsystems, Inc., Palo Alto, California, 1999. Available at: http://www.sun.com/jini/specs/jini1_1spec.html (21st of December 1999).

[Sun99c] Sun Microsystems, Inc.: *Jini Technology Specifications*. Sun Microsystems, Inc., Palo Alto, California, 1999. Available at: http://www.sun.com/jini/specs (21st of December 1999).

[Sun99d] Sun Microsystems, Inc.: *Jini Technology 1.0.1 API Documentation*. Internet WWW-page, URL: http://developer.java.sun.com/developer/products/jini/jinidoc.html (21st of December 1999).

[Sun99e] Sun Microsystems, Inc.: *Enterprise JavaBeans Specification 1.1*. Sun Microsystems, Inc., Palo Alto, California, 1999. Available at: http://java.sun.com/products/ejb/docs.html (10th of August 2000).

[Sun99f] Sun Microsystems, Inc.: *Java 2 Platform, Standard Edition, v1.2.2 API Specification*. Internet WWW-page, URL: http://java.sun.com/products/jdk/1.2/docs/api/index.html (1st of August 2000).

[Sun99g] Sun Microsystems, Inc.: *EmbeddedJava Application Environment Specification 1.1*. Sun Microsystems, Inc., Palo Alto, California, 1999. Available at: http://java.sun.com/products/embeddedjava/spec/index.html (30th of May 2000).

[Sun99h] Sun Microsystems, Inc.: *PersonalJava Application Environment Specification 1.1.2*. Sun Microsystems, Inc., Palo Alto, California, 1999.

Available at: http://java.sun.com/products/personaljava/spec-1-1-2/index.html (30th of May 2000).

[Sun99i] Sun Microsystems, Inc.: *Jini Technology and Emerging Network Technologies*. A white paper, Sun Microsystems, Inc., 1999. Available at: http://www.sun.com/jini/whitepapers/technologies.pdf (7th of August 2000).

[Sun99j] Sun Microsystems, Inc.: *The K Virtual Machine (KVM)*. A white paper. Sun Microsystems, Inc., Palo Alto, California, 1999.

[Sun00a] Sun Microsystems, Inc.: *Connected, Limited Device Configuration Specification Version 1.0, Java 2 Platform Micro Edition*. Sun Microsystems, Inc., Palo Alto, California, 2000.

[Sun00b] Sun Microsystems, Inc.: *Mobile Information Device Profile (JSR-37) JCP Specification, Java 2 Platform, Micro Edition, 1.0*. Sun Microsystems, Inc., Palo Alto, California, 2000.

[Sun00c] Sun Microsystems, Inc.: *Jini Technology Surrogate Architecture Specification*. Sun Microsystems, Inc., Palo Alto, California, 2000.

[Sun00d] Sun Microsystems, Inc.: *Java Card 2.1.1 Application Programming Interface*. Sun Microsystems, Inc., Palo Alto, California, 2000.

[Swa99] Swainston-Rainford, M.: Serving up Beans to the Enterprice. *Application Development Advisor*, 2(4): 59-61, 1999.

[Sys00] Systä, K.: *JINI for mobile terminals with Bluetooth (BT) connectivity*. A technical white paper, Nokia Research Center, 2000.

[Thi98] Thiruvathukal, G. K., Thomas, L. S. and Korczynski, A. T.: *Reflective Remote Method Invocation*. ACM 1998 Workshop on Java for High-Performance Network Computing, ACM, 1998.

[W3C97] W3C: *Extensible Markup Language (XML)*. W3C Proposed Recommendation PR-xml-971208, W3C, 1997. Available at http://www.w3c.org/TR/PR-xml-971208 (10th of December 2000).

[Wal94] Waldo, J. & al.: *A Note on Distributed Computing*. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Mountain View, California, 1994.

[Wal99] Waldo, J.: The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7): 76-82, 1999.

[Wal00] Waldo, J.: Alive and Well: Jini Technology Today. *Computer*, 33(6): 51-56, 2000.

[Wei96] Weiser, M.: *Ubiquitous Computing*. Internet WWW-page, URL: http://www.ubiq.com/hypertext/weiser/UbiHome.html (9th of June 2000).

[Wel99] Wells, D. & al.: *Software Survivability*. Proceedings of the DARPA Information Survivability Conference and Exposition, 2000 (DISCEX ’00), IEEE Computer Society, 1999, 241-255 (vol. 2).

[Won98] Wong, H.: What's all the hype about Enterprise Java Beans? *Java Report*, 3(7): 62-64, 1998.

[WuD98] Wu, D., Agrawal, D. and Abbadi, A. E.: *StratOSphere: Mobile Processing of Distributed Objects in Java*. Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, ACM, 1998, 121-132.