University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

8-2000

JiniOS : the network is the computer

Richard Clippard

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation

Clippard, Richard, "JiniOS : the network is the computer. " Master's Thesis, University of Tennessee, 2000.
https://trace.tennessee.edu/utk_gradthes/9326

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Richard Clippard entitled "JiniOS : the network is the computer." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce Whitehead, Major Professor

We have read this thesis and recommend its acceptance:

Dinish P. Mehta, K. R. Kimble

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council

I am submitting herewith a thesis written by Richard Clippard entitled "JiniOS The Network is the Computer" I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science

[illegible]

Bruce Whitehead, Major Professor

We have read this thesis
And recommend acceptance

[illegible]

[illegible]

Accepted for the Council

[illegible]

Associate Vice Chancellor
And Dean of the Graduate School

JiniOS: The Network is the Computer

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Richard Clippard
August 2000

No liability is assumed regarding the use of the information contained herein, and Richard Clippard or the University of Tennessee Space Institute (UTSI) assumes no responsibility for errors or omissions, neither is any liability assumed for damages resulting from the use of the information contained herein

Zip is a registered trademark of Iomega, Inc
Pentium is a registered trademark of Intel Corporation, Inc
Inferno is a registered trademark of Lucent Technologies, Inc
iButton is a registered trademark of Dallas Semiconductor, Inc
Java, RMI, Solaris, "The Network is the Computer", and Jini are registered trademarks of Sun Microsystems, Inc
Windows NT, Windows 95/98, Plug-and-Play are trademarks of Microsoft Corporation

All other brand names and product names mentioned in this document and in this software are trademarks, registered trademarks, or service marks of their respective owners

Dedication

This thesis is dedicated to my parents, Gisela and Garnett, and to my wife, Pamela, for her encouragement, support, and endless patience

Acknowledgements

I want to thank my advisor and committee chairman, Dr Whitehead, and the other members of my committee, Dr. Mehta and Dr Kimble, for their instruction, guidance and assistance. I want to thank my brother for providing me with the computers used in my research and all his technical assistance I also want to thank Sandra Teal Reed for her reviews and inputs I would like to thank Keith Edwards, author of the book Core Jini, for helping me with several Jini programming problems even though his book was already done and Sun Microsystems for realizing the power of releasing new products as Open Source so that everyone can have equal access Lastly, I would like to express my gratitude to those people who reviewed by paper and to those who listened patiently as I preached

Abstract

While computers have grown more powerful and the operating systems that drive them have become easier to use, the process for installing new hardware has changed very little from the techniques used with the earliest computers The current method for installing new devices into a computer is much too complicated even for an experienced user and often results in hours of frustration. In spite of the fact that there has been phenomenal growth both in the speed and complexity of computers and the peripherals attached to them, the process for installing these devices remains user intensive This research addresses some of the shortcomings of the current process and defines an operating system called JiniOS, which implements the solutions proposed herein JiniOS allows devices to install themselves into any computer only when they are needed, without any setup and very little user intervention

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Michael Dertouzos, the director of the M I T Laboratory for Computer Science for the past 25 years relates the following story

> Last year a few of us from the Laboratory for Computer Science at the Massachusetts Institute of Technology were flying to Taiwan I had been trying for about three hours to make my new laptop work with one of those cards you plug in to download your calendar But when the card software was happy, the operating system complained, and vices versa Frustrated, I turned to Tim Berners-Lee sitting next to me, who graciously offered to assist After about an hour, the inventor of the Web admitted that the task was beyond his capabilities
>
> Next I asked Ronald Rivest, the co-inventor of RSA public-key cryptography, for his help Exhibiting his wisdom, he politely declined At this point, one of our youngest faculty members spoke up "You guys are too old Let me do it " But he also gave up after and hour and a half So I went back to my "expert" approach of typing random entries into the various wizards and lizards that kept popping up on the screen until by sheer accident, I made it work three hours later (Dertouzos, p52)

Clearly, these were not people new to computers and this task, installing a card made for the device, should have been trivial However, all too often when dealing with computers, what should be simple turns out to be almost impossible This quote is from an article discussing M I T 's new infrastructure for information technologies, the Oxygen system - a hardware and software system for handling information technology He outlines three goals to allow people to "do more by doing less" bring technology into our lives, increase human productivity and ease of use, and offer these gains to all Of the three, the key is increasing the ease of use (Dertouzos)

"Things should be made as simple as possible, but not any simpler" was a sentiment often echoed by Albert Einstein, but this seems to go contrary to the current world of computers and computer technology Despite massive efforts on the parts of hardware and software manufacturers, computers are still too difficult for the average user to configure and maintain

> For the first 40 years of computer science, we have been preoccupied with catering our technology to what machines want We design systems and subsystems individually and then throw them at the public, expecting people to make different components work together (Dertouzos, p52)

The market is fragmented into various camps such as personal computer (PC), Apple, and Unix, containing components that are not interchangeable A computer is typically composed of dedicated devices such as CPU, memory, keyboard, mouse, monitor, and hard drives, which must be installed and configured for use in the intended operating system for Once configured, the makeup of the computer changes very little for the life of the system In addition to the complexity of the basic system, an experienced system administrator must handle setting up and maintaining a network by installing the proper drivers and setting up the configuration files Sharing devices, such as hard drives and printers, in a homogenous system of computers can be difficult and almost impossible in a heterogeneous system This leads to duplication of basic devices such as printers, tape drives, and hard drives

The personal computer is too complex, but the technology it represents is not necessarily more complex than many common household items the television, microwave and stereo for example But while we spend countless hours maintaining our PC's, we spend hardly any time maintaining these other systems which raises the question "Why are computers so complex?" The PC's complexity can be attributed to three things the attempt to make a single device to too many things, the need to have a single machine suffice for every person in the world, and the business model of the computer industry has to convince the consumer that he needs a new version of a software package or a new computer This complexity stems out of the desire to make the same personal computer be all things to all people (Norman)

Donald Norman states that the solution to the computer complexity problem is to separate the computer into "information appliances" which he defines as

> appliance n
> A device or instrument designed to perform a specific function, especially an electrical device, such as a toaster, for household use
>
> information appliance n
> An appliance specializing in information knowledge, facts, graphics, images, video, or sound An information appliance is designed to perform a specific activity, such as music, photography, or writing A distinguishing feature of information appliances is the ability to share information among themselves (Norman, p 53)

A tool should be designed for each specific task so that its form, features and structure make it a perfect fit Each tool should be designed so that the complexity of the appliance is the task not the tool, to allow interaction, and to be fun to use The goal is to make each tool match a single and specific activity

If we carry the idea of an information appliance to today's systems and separate the basic computer into individual devices containing enough intelligence to communicate their specific requirements to the operating system, we can go a long way to simplifying some of the complexity inherent in personal computers In such a distributed system each new component announces its presence to the operating system and provides the necessary information to use it only if requested Sun Microsystems has proposed the Jini library as a software mechanism to support this kind of distributed hardware over a network Jini consists of a small amount of Java code that resides on each device and a set of conventions to create a "federation" of Java virtual machines on the network Federation participants are dynamically connected to share information and perform tasks

This idea of shared devices is not new, nor is it specific to Jini All of the major operating systems provide some mechanism to export some of the components contained within the computer However currently the operating system into which the device is installed has to know how the device will be used This typically involves installing a vendor-supplied driver for the specific operating system, which can be difficult to do and places the entire operating system overhead on the host desktop

One of the fundamental reasons that computers are too complex has to do with the relationship between the computer and its components The personal computer of today is just a miniature version of the large mainframe of yesterday with all of its problems and idiosyncrasies The computer still views all of the devices installed in it, or attached to it, as slaves and the devices view the computer as their master This single concept can be blamed for most of the problems encountered installing hardware and for the fragmentation of hardware that exists between operating systems today The current techniques for installing, accessing, and maintaining hardware are outdated and must be redesigned and the hardware inside the computer must be replaced by external, self-configuring networks of devices

I propose, as an example of such a computer, to create a federation of networked devices using Jini and a minimum of four personal computers that have been networked via an Ethernet connection One computer will be the main system and all others will export their respective devices (monitor, keyboard, mouse, hard drives, and printer) making themselves available to the main system While devices will reside inside the computer case, as in the case of the hard drive, or be connected to it, as with the monitor, they will be treated as independent entities The intent is to simulate independent devices using the power and intelligence of the personal computers containing them The inclusion of new devices into the federation will be simulated by adding a personal computer and all of its devices to the network Each device in the newly added computer will announce itself and become available for use by the main computer

As a point of reference, the configuration for this project involved installing and configuring a hard drive, a floppy drive, a CD-ROM drive, and a Zip drive into a personal computer Also network cards were installed into each of the four PC's which were then connected to a hub and networked via the Windows 95 operating system This task required two programmers with a combined experience of over twenty years over eight hours to complete This task should have taken 10 minutes

Of course, in a truly distributed system each device would be independent and separate, containing a power source and an Ethernet connection, however, that is beyond the scope of the project The intelligent devices containing even the small amount of processing power and memory and an Ethernet connection required for this project are not yet available

Jini is still early in its development cycle and not intended for the general public This project will entail not only getting the basic Jini software system operational but also writing drivers for the various hardware components The goal is to demonstrate a computer that is composed of a system of distributed hardware components tied together using Jini

Chapter 2
Review of Current Operating Systems

2.1 Hardware

The Microsoft Windows operating systems (both NT and Windows 9x and 2000) as well as the underlying Disk Operating System (DOS) rely on the use of device drivers to access the various devices connected to a personal computer The device driver converts the more general instructions of the operating system to specific messages that the device type can understand The keyboard, mouse, hard drives and CD-ROM are examples of devices that are accessed using device drivers While the drivers for the more typical devices, such as hard drives are included in the operating system, newer devices use drivers provided by the device manufacturer and are installed into the operating system when the device is connected to the computer Additionally, the manufacturer may supply software, in the form of Dynamic Link Libraries (DLL's) or information files, which facilitate the communication between the user software and the operating system devices For example, a printer uses several other programs to setup the printer, format the text and transfer the final stream to the printer as well as countless files that contain font definitions All the device drivers and ancillary files must be stored on the computer before the device can be used In a typical installation, the device drivers and their associated libraries account for one megabyte of space per printer and fourteen megabytes for fonts

The device driver communicates with a device via the computer's basic input/output system (BIOS) which is loaded from static memory when the computer is first turned on The average BIOS for a personal computer contains about one million lines of low-level code (Pinto) It is initially responsible for performing a "power-on self test" (POST) to determine the health of the components, checking the system configuration to determined how to finish the boot process and loading the boot record which eventually loads the operating system After the operating system has taken over, the BIOS remains as the conduit from the driver, installed in the operating system, to the actual hardware device. The BIOS assigns and manages the interrupts from the processor when something happens to an installed device For example, when a key is pressed the processor performs an interrupt that is handled by the BIOS to read the key. In addition, the BIOS may transfer data to the operating system by storing it in a predefined region of memory known as the base address Typically, each installed device has its own interrupt and base address that are specified when the device is installed ("What is BIOS") (Risley)

Devices are defined to the BIOS either by direct user specification during BIOS setup when they are first installed in the computer, or through a mechanism known as 'Plug and Play' in which the device identifies itself to the system when the computer is booted Since several devices are stored in the computer, the computer must be powered off to install them Moreover, since changes to the BIOS only take effect when the computer is first booted, it must be powered off and back on to access new external devices Devices are defined to the operating system by its device drivers If the operating system does not already contain a driver for the device being installed, one must be installed before it can be used Oftentimes, the computer must be restarted to load these drivers into the operating system

2.2 Evolution

> the basic structure of our computers remains much the same as it was in the 1950s We have central processing units, memory, and disks And, despite the fact that we now use computers with power that would have been unimaginable 40 years ago, to play games and balance our checkbooks, we fundamentally interact with these machines in the same ways that our predecessors did We install software on them, run applications, and manage the (always scarce) disk resources of our systems A mainframe systems administrator from 1950 would understand these tasks immediately In fact, in some sense, we've *all* become systems administrators (Edwards, p 5)

Operating systems evolved to this structure because of an assumption early on that processors and memory were either expensive or not available or both, and that the fastest and most reliable way to communicate is through a direct internal connection The idea was to have a single computer attached to dumb devices which relied on the computer to manage and communicate with them The computer would know everything about all the devices attached to it and all devices would follow a specified communications protocol Hardware designers rely on the processing power of the personal computer to push information to the device The keyboard communicates via the serial port, the printer via the parallel port, the mouse via the serial port or bus, and the internal devices such as hard drives, Zip drives and CD-ROM's via the internal bus In all cases, the device relies on the computer to relay the information to it via a dedicated and specialized connection The operating system is the centralized repository of all the system's information on users, registered machines and addresses which means it tends to grow quite large—as much as 18 2 million lines of code for Windows NT and Solaris (Pinto) In addition, when PCs are connected to networks, system administration and maintenance costs dominate the total cost of ownership (Schmidt)

When DOS was released in 1982, the state of the art processor for the PC was the Intel 8080 Over time processor speeds increased and prices for CPUs and memory dropped and computing power became much more readily available DOS evolved into Windows 2 0 and eventually Windows 95, but the mechanism for handling devices was never revised, partially because of the massive investment by both manufacturers and consumers in dumb devices, and partially because it seemed to work Later when Linux was released for the PC there was a strong desire to use standard devices already installed into personal computers so Linux had to implement this same scheme for accessing these devices, at least at the lowest level

This arrangement of requiring the operating system to have knowledge of all possible devices that may be connected to it worked well while the systems that people used were relatively homogeneous and static and the choices of devices were limited It did however, make installation of devices much more difficult, because not only did the installers have to connect the hardware to the computer correctly, they also had to connect the device to the operating system as well Additionally as the number of possible devices has grown, so has the number of device drivers installed in the operating system As the number of choices has grown the process for installing new devices seems to have become even more difficult

For example, the Zip drive from Iomega can be either internal or external, and connected to either the parallel port, SCSI port or the internal bus Each of these is mutually exclusive, i e , they are not interchangeable, and the user must decide which one to purchase depending on the requirements and availability The driver that is installed depends on the connection and operating system Drivers that work with Windows 95 will not work with Window NT or Linux Once the drive and driver are installed, additional software is installed so that the operating system treats the Zip drive as a standard drive It is easy to use from this point having been incorporated into the operating system, but the installation process itself can be somewhat daunting

2.3 Innovation

This process must be completed for each new device added to an existing system Sometimes the system must be rebooted before the additions take effect Each new device must be configured to work in the existing computer with the existing operating system While Microsoft has worked extensively to make the process of device configuration easier, it can still be very frustrating even for experienced users One improvement is "Plug and Play," which allows the device to identify itself and suggest possible configurations, and allows the operating system to organize the devices to avoid conflicts However the device is still viewed as a dumb piece of hardware, and all the device drivers must be pre-installed for the device to work It also does not simplify the process of physically installing the hardware into the computer system

As pointed out before, operating systems access hardware devices through loadable device drivers All modern operating systems define their own interface, kernel device model, between the operating system and the device drivers thereby keeping the operating system free from device details Different drivers must be designed for all operating systems accessing a particular device because how operating systems access a device via the driver is different for each operating system A possible solution to this problem is to develop a portable device driver module, called a hardware device module, which resides between the kernel device module and the device The hardware device module is written independent of any specific kernel and is meant to model the particular hardware device that the driver supports The kernel device module adapts the device-specific interface of the hardware device module to the general device model of a particular operating system Since the hardware device module is written independent of any specific kernel, it only has to be designed once for each device Each driver still has to be distributed according to the target operating system, but the task of converting the hardware device module to another system should just require that it be compiled for that system (Ryan) This does not address the issue of installation but it makes the task of writing device drivers much easier and would make devices more readily available across different operating systems

Another innovation that does address the installation problem is the Universal Serial Bus (USB) USB allows the user to chain together several external devices and connect a single wire to the computer, similar to the Small Computer System Interface (SCSI) Each device in the chain is assigned an identification number that is used to address that particular device USB devices are "hot swappable," that is, devices may be added to or removed from the chain without restarting the computer system because they identify themselves to the operating system when they are connected USB is currently limited to 12 megabit (Mbit) throughput shared by all devices and a chain length of five meters with 127 devices and up to five hubs Device drivers for classes of devices, such as keyboard or camera, still have to be installed in the operating system before the device can be used, however, devices are more easily added to an existing system (Quinnell) Given the limited length and the fact that some drivers still have to be installed, USB is not the final solution either

2.4 Solution

Regardless of the device or the connection, the requirements of the operating system have remained the same The operating system is expected to know everything about the device before it can be used The process of installation is greatly simplified because the device is external so the case does not have to be opened and because it no longer depends on internal settings such as base address and interrupt number, however, the user must still install an additional piece of software into the operating system before the device can be used. A much more palatable solution would be one in which the device explains to the operating system how the device is accessed and used This type of system would enlist a single common means of communication, as well as a common language so that the operating system and the device can initiate communication This is similar to how the BIOS operates in the personal computer, except in this case the device would be able to upload its specifications and access mechanisms Finally there must exist a means to discover new devices and determine when previously accessed devices are no longer available

The hardware must be modıfied so that the devıce has enough processıng power to communıcate effectıvely wıth the host computer, and has enough statıc memory to store configuratıon ınformatıon about ıtself on the devıce so that ıt can be sent to the operatıng system when necessary Addıtıonally each devıce must have the abılıty to communıcate vıa the establıshed lıne Thıs would requıre both a standardızed physıcal connectıon as well as a standardızed protocol to serve as a basıs for communıcatıon Addıtıonal hardware changes would be necessary to enclose the devıce and provıde power Thıs ıs already the case for a majorıty of standard perıpheral devıces, such as prınters and scanners and, in others, the push for USB ıs dıctatıng the externalızatıon of devıces

Havıng all devıces external to the computer allows system configuratıon changes to be made rapıdly and easıly by merely connectıng or dısconnectıng devıces to the system The operatıng system then must determıne the devıce's avaılabılıty, and how each devıce ıs used as ıt ıs connected The computer system becomes a combınatıon of dıstrıbuted components, the number and configuratıon of whıch can grow or shrınk as requırements change No specıal skılls are requıred to make system configuratıon changes Such a system would then be called a dıstrıbuted computer system

Ideally, whether the devıce can be used ın a computer should be ındependent of the type of computer or ıts operatıng system All devıces should be usable by all computers Thıs means not just personal computers, but also Apple computers, workstatıons and maınframe computers The proprıetary aspects of the devıce should be hıdden from the operatıng system and especıally from the user

The best computer operatıng system ıs one that ıs transparent to the user It allows you to change the configuratıon of a system without havıng to know anythıng about computers, and wıthout havıng to dırectly modıfy the operatıng system by ınstallıng addıtıonal drıvers to access newly connected devıces For example, the home stereo system does not care what brand of component ıs attached to the amplıfier The system software does not have to be modıfied or rebooted to change the CD player Except ın the case of very hıgh-end components, the users does not have to worry about the brand or types of components or connectıons they have when decıdıng to add another devıce A more computer related example ıs that of the Musical Instrument Data Interchange (MIDI) standard that allows for the sharing of musical information among devices As long as the device follows the MIDI standard, it can be used without having to worry about compatibility issues

> A keyboard can connect to a synthesizer, rhythm machines, control pedals, and other instruments, creating orchestras that unleash the power of digital recording and synthesis New instruments can be invented, such as a voice device that allows singing to be converted to MIDI sequences that are then fully mixed with other devices
> (Edwards, p 58)

Chapter 3
Distributed Systems

3.1 Introduction

> Because each workstation has private data, each must be administered separately, maintenance is difficult to centralize The machines are replaced every couple of years to take advantage of technological improvements, rendering the hardware obsolete often before is has been paid for Most telling, a workstation is a large self-contained system, not specialized to any particular task, too slow and I/O-bound for fast compilation, too expensive to be used just to run a window system For our purposes, primarily software development, it seemed that an approach based on distributed specialization rather then compromise could better address the issues of cost-effectiveness, maintenance, performance, reliability, and security (Schmidt, p 32)

There are a number of advantageous features of distributed operating systems that have motivated research in the area of distributed systems for the past forty years These include resource sharing, performance, reliability, availability, price, extensibility, and network transparency (Mull) Distributed systems have typically worked well for highly specialized applications They tended to focus on allowing computer systems either to share resources or to distribute the load of certain intensive operations Operating Systems that allow distribution of some resources are common Windows 95 for example allows certain specific devices such as hard drives to be shared among computers Similarly, Network File System (NFS) allows drives to be shared in the Unix operating system In both cases, the owner of the device exports it (either to a specific user or for global access) and the user desiring to use it must mount it before accessing it. In both cases, the exporting computer must be a complete system, i e , each method relies on software, operating system and possibly other devices to manage the device sharing. Both methods are mutually exclusive—Windows does not allow devices that have been exported via NFS to be mounted without the addition of specialized third-party software, and Unix systems will not typically NFS mount devices exported from a Windows system

Distributed systems also have not typically operated well in a heterogeneous system of computers, or have required specialized programming to take advantage of available devices They have not been at the component level, but rather between complete systems Finally, they still did not address the connectivity problem, that is, they did not address the physical connection between the computer and its installed devices They did not address the fact that certain devices do not work with certain operating systems or computers because of either proprietary connections or software

3.2 Survey of Distributed Systems

Plan 9 from Bell Labs, for example, is a distributed operating system written with the goal of building a time-sharing system out of workstations It includes a new operating system (loosely based on Unix), network protocol and compilers Plan 9 gives users a personal view of the public space by using local names for globally accessible resources (Pike) Inferno is another new network operating system written to "deliver content in a rich environment of heterogeneous networks, clients and servers" ("Inferno", p. 1) It is also from Bell Labs and, like Plan 9, it organizes resources in a hierarchical file system, uses private and global name spaces, is built on a common communication protocol, handles security issues and has its own programming language (Limbo) and reference API's. Unlike Plan 9 however, Inferno was designed to run either as either the native operating system or hosted by another operating system ("Inferno la Commedia Interattiva") Neither system provides an automatic method for installing and maintaining connected devices

The JINOS (Pinto) operating system is based on Java and Jini and is also a complete replacement of the operating system The approach is to supply only the computing power necessary to run the Virtual Machine effectively and to let the user build an operating system customized to his requirements by starting with a processing module and adding services as needed The architecture is composed of a hardware layer, a microkernel layer, a hardware abstraction layer and the Virtual Machine

The NetSolve (Plank) software environment for networked computing uses a client-server model that allows users to distribute a variety of computational resources Clients specify the computations that they want done by contacting agents who maintain information on server availability, load, and support software The agents then route the request to servers to perform the computations Two levels of fault-tolerance are employed—on the server in individual computations and using agents that detect server failures by a time-out mechanism and move computations to other servers

The desk area network (DAN) (Barham) is an architecture for the internal connection of a computer system using asynchronous transfer mode (ATM) as the interconnect between components Every device is equipped with an ATM interconnect instead of the standard computer bus interface Devices are classified into three categories, dumb, supervised and smart, according to their processing power and ability to perform local management While the DAN is intended to be the internal interconnection of a machine, the current operating system is still responsible for access control and protection (Hayter)

WebOS (Vahdat) is a framework for supporting applications that are geographically distributed Its goal is to provide a common set of operating system services to wide area applications and make wide area resources as easy to use as those available on a LAN or local resources It includes mechanisms for resource discovery, global namespace, remote process execution, resource management, authentication and security. The intent is to write applications based on the WebOS framework to access remote services

Similarly, JetFile (Gronvall) is a distributed file system designed for a heterogeneous environment such as the Internet It efficiently handles daily tasks such as mail processing and document preparation with the intent that a typical user will have no more incentive to store files locally than in the JetFile system It relies on "peer-to-peer" communication over multicast channels to facilitate resource location and retrieval JetFile employs leasing and callback mechanisms to maintain the coherency of the system A callback is a notification that a cached item is no longer valid In the event that the callback is lost, clients will eventually discover the service is no longer available when their requests time out

The Stateless, Low-level Interface Machine (SLIM) (Schmidt) architecture defines a desktop machine as a simple, stateless, I/O device that uses a dedicated, off-the-shelf interconnection technology to access a pool of computational resources The SLIM system was designed around low-level hardware (display, keyboard, mouse, etc) and a software-independent, dedicated 100 megabits per second switch Ethernet protocol In this way the advantages of thin-client architectures, such as resource sharing, centralized administration, and reduced cost, are maximized A prototype SLIM system was used by over 60 engineers, managers, marketing personnel, and support staff "as their only desktop computing device for the past year, and they have found their interactive experience to be indistinguishable from that of working on high-end workstation-class machines" (Schmidt, p 34)

3.3 Common Attributes

JiniOS is different from the classical distributed operating system, which seem to be focussed more on distributing the processing power (CPU's) over a computational task The central functionality that JiniOS focuses on is that of network-attached peripherals where a computer peripheral communicates via a network rather than a traditional I/O bus (Van Meter) Unlike other systems, where the entire operating system was redesigned to support distributing all services, in JiniOS only the peripherals are distributed and every attempt was made to design a system that could be incorporated into existing operating systems Additionally the intent was to accomplish this using existing technology

A system using Network Attached Peripherals (NAPs) is a heterogeneous distributed system and the existing body of research on such issues as discovery, naming, deadlock, and security are all relevant (Van Meter) The protocols and mechanisms used by a network-connected device to become aware of the network to which it is connected and the services available to it is called discovery The discovery process usually involves a server at a known address that tracks services as they are announced or a distributed server system that employs multicasting to identify servers and services Discovery usually necessitates some form of naming system to describe services so clients may choose the appropriate service This naming system must be expressive enough to fully describe a wide variety of services, based on specific properties of services, responsive, able to handle failures and network topology changes, and easily configurable

The importance of resource discovery and service location is only now receiving attention in the research community The Intentional Naming System (INS) (Adjie-Winoto) was written to support dynamic networks of mobile computers and devices In this system, a client uses a name to request a service without explicitly specifying the end node INS uses a decentralized network of "resolvers" to discover names and route messages based on advertised capabilities and client requirements These resolvers self-configure into a spanning-tree, application-level overlay network that clients use to resolve their requests and advertise services It implements names using expressions that are broken into attribute, the category in which an object is classified, and value, the object's classification Services periodically advertise their names to the INR system on a well-know port As long as services are refreshed, they are kept alive, however, services that have not been refreshed after a pre-defined time (lifetime), are discarded The current implementation of INR is written in Java to take advantage of its cross-platform portability, and INRs use UDP to communicate with each other

Most of the techniques being researched in the area of networked devices employ some sort of leasing to determine the health of the network and the availability of services Services register and periodically renew with a server or client If the lease lapses without a renewal, the service is dropped from a list of viable candidates

> The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central manager Partial failure is a central reality in distributed computing Both the local and the distributed world contain components that are subject to periodic failure In the case of local computing, such failures are either total, affecting all the entities that are working together in an application, or detectable by some central resource allocator (such as the operating system on the local machine) This is not the case in distributed computing, where one component (machine, network link) can fail while the others continue . In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that line ("Discovery Devices and Services in Home Networking", p 5)

In order for the networked device technology to move from the research environment to the home, it must be accepted by the consumer In order to be accepted it must be easy to use Consumers will overlook installation problems if the product is easy to use but will not accept a product that is easy to install but hard to use Furthermore, the networks that support these devices must be self-configuring and transparent to the user ("Discovery Devices and Services in Home Networking") This has not been the case with distributed systems in the past since they were not targeted at the general user

Chapter 4
Concepts Used to Develop JiniOS

4.1 Introduction

JiniOS represents an operating system that addresses the shortcomings of how current operating systems install and access devices It deals specifically with this aspect of the operating system—it does not attempt to supplant the entire operating system The concepts and techniques used in JiniOS can be retrofitted into existing operating systems without having to totally rewrite them This is an important feature, and one that may help these concepts to become universally accepted The focus of this research is to determine if the tools to build such an operating system exist While the discussion will be focused primarily on the personal computer world, there is nothing in its design to exclude it from being used in the workstation or even mainframe systems In fact devices that are configured to work with JiniOS will work regardless of the primary operating system or even underlying chip set (Intel Pentium, Mips, Alpha, Sun Motorola)

The JiniOS host runs a server that registers all devices that answer requests for services All initial communication between the client computer and between the server and the devices is over a standard Ethernet line using standard Internet protocols Each device stores all the software necessary to communicate with it which it uploads to the server when the device registers itself with a server When a computer wants to use the device for the first time, it polls the server for devices of a particular type that have been registered Once a device has been selected, the software registered by the device is then downloaded from the server to the client computer and the client then begins communication with the device using the downloaded software Communication with server is terminated until the next request, so in essence the device driver is installed only when necessary (Figure 4 1) This means that the computer must be able to run the software loaded from the device regardless of the type of device or the type of computer To date the only language for which this is true is the Java programming language, therefore, the initial software loaded from the device must be written in Java This does not mean that all the software on the device must be written in Java, but rather that software that will run on another computer must be in Java

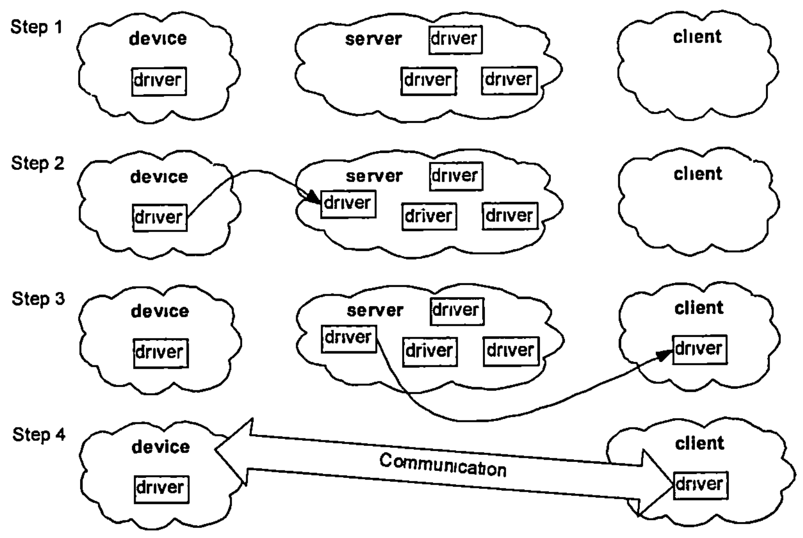


Figure 4 1 Basic client/server model

JiniOS connects all "smart" devices to the core system through standard Ethernet connections Because each device must be capable of independent Ethernet communication, it must have an Ethernet address, an Ethernet connector and some small processor and memory that can handle Ethernet communication Each device must have its own power source, and, lastly, each device must be capable of running a Java Virtual Machine (JVM) in addition to any other software required for client communication The ability to run a JVM is required because JiniOS is based on Jini, the Sun Microsystems library of Java classes for distributed applications

Another class of languages that offer the same "write-once-run-anywhere" capability as Java are the scripting languages such as Tcl and Perl Like Java, programs written in one of these languages can be shipped to another computer and run there, as long as the interpreter is available on the target machine However, while these offer similar portability of source code, they are not as well suited as Java because they do not provide the object-oriented constructs available in Java, nor can they enforce Java's level of security restrictions Virtual machines, the JVM for example, provide a platform-independent binary format, security features for executing untrusted code, and an extensive set of programming interfaces that embody those of a general-purpose operating system Future networks will be characterized by mobile code, large numbers of hosts, and large numbers of devices per user that span different operating systems and hardware platforms Virtual machines have a potential to play a large role in these networks (Sirer) The use of virtual machines turns an otherwise heterogeneous assortment of computers into a homogenous collection of seemingly identical machines (Waldo, 1994)

4.2 Ethernet

Communication on the Internet is comprised of a set of protocols that grew out of the United States Defense Advanced Research Projects Agency (DARTA) desire for a more reliable communication protocol for is packet-switched wide-area network known as ARPANET The Internet Protocol (IP), the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are the most widely used protocols in the suite named TCP/IP that makes up today's Internet TCP provides reliable transmission, ensuring error-free transmission of data by providing acknowledgement of packets received, retransmission of lost packets, and reordering of out of order packets, but it ties up system resources to do so UDP on the other hand does not guarantee either that the packets are delivered or in what order they will be received, but it consumes very little system resources TCP is appropriate for connections between computers where a continuous connection is important (telnet) or large amounts of data are being transmitted (ftp) UDP is best suited to situations where a host is monitoring a port that may be accessed by multiple hosts that request small bits of data (http) Both protocols wrap the data to be transferred with additional host and destination information IP is responsible for transmission of data across networks by wrapping TCP or UDP packets with additional information necessary for the delivery of the data Protocols above TCP and UDP such as Telnet, FTP and NFS add another layer to the packets before the TCP or UDP layer. Essentially one could think of a transmitted packet as being stored in a series of smaller boxes, each contained in the next larger one (Figure 4 2) The outer box is the IP layer, then the TCP or UDP layer, and finally the Telnet or FTP layer

The current IP header standard, called IPV4, is being revised to IPV6 in order to overcome some of the shortcomings in the current standard One of these shortcomings concerns the fact that we are running out of addresses because of the way they are assigned and because the number of networked machines is growing very rapidly The current standard is based on 32-bit address which allows for over four billion

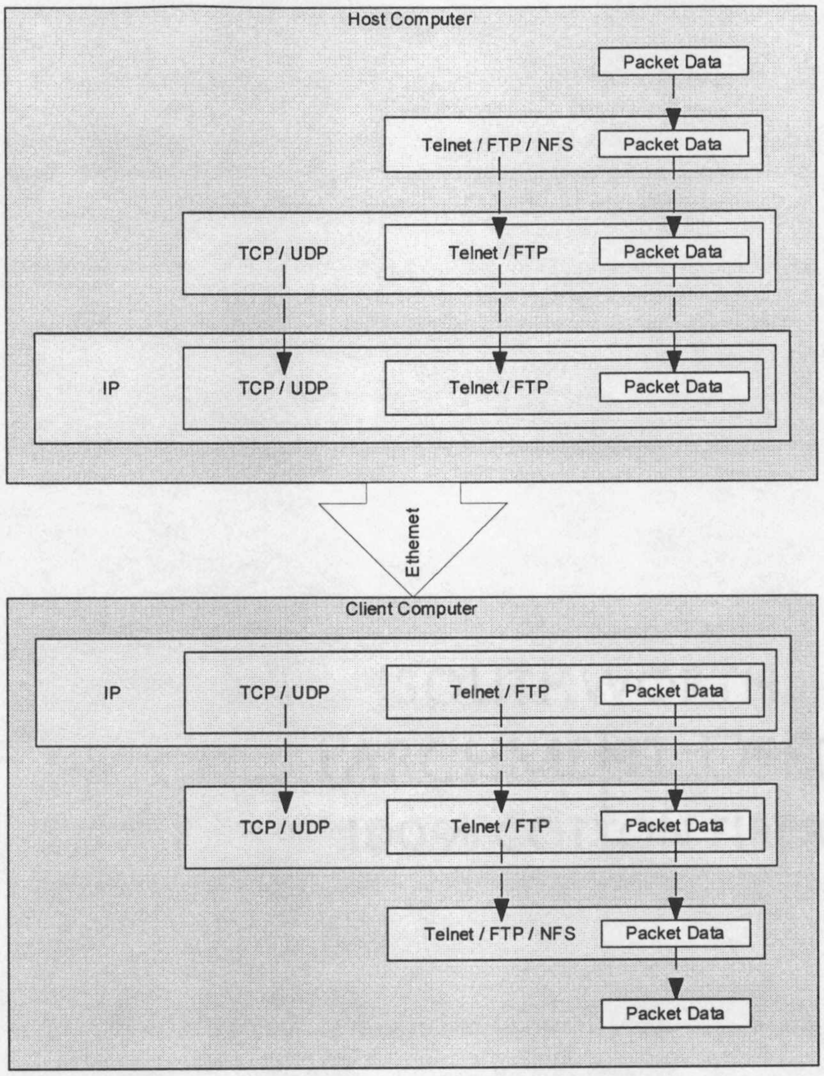


Figure 4.2. Ethernet packet layers.

address (2^{32} = 4 3 bıllıon), but IPV6 ıs based on 128-bıt addressıng whıch allows for 3 4x10^{38} addresses (2^{128} = 3 4x10^{38}) (Montfort)

An Ethernet card or software that emulates a card, such as that used by a modem, ıs requıred to communıcate vıa TCP/IP The host software transmıts and receıves the IP packets usıng thıs card connected to a network vıa a standard class 5 (thın wıre) connector that looks lıke a bıgger versıon of a phone connector (called an RJ45 connector) Current versıon of Ethernet protocol are generally lımıted to eıther 10 megabıts per second (Mbps) or 100 Mbps but there ıs nothıng ın the protocol ıtself that sets an upper lımıt Each card, regardless of manufacturer or date of manufacturer, contaıns a unıque ıdentıfıcatıon number known as ıts Ethernet address Thıs number ıs mapped to 32-bıt number known as the Internet or IP address Thıs number, famılıar to most users, takes the form 192 168 2 1, where each dotted sectıon or octet represents another level of address refinement startıng wıth the leftmost set

Internet addresses are assıgned by the Network Informatıon Center (NIC) located at SRI Internatıonal. IP addresses ın the Western Hemısphere and sub-Saharan Afrıca are assıgned by NIC, and they get the numbers from the Amerıcan Regıstry for Internet Numbers European IP addresses come from the Réseux IP Européens, and Asıan IP addresses come from the Asıa Pacıfic Network Informatıon Center

Today, early 2000, the Internet contaıns mıllıons of computers all runnıng the same standardized TCP/IP protocol, regardless of the type of computer ıt is or the operatıng system runnıng on ıt Dıfferent programs may add addıtıonal layers under the basıc IP, but IP ıs used to delıver the packets. The growth of the Internet can be attrıbuted to the standards establıshed by the TCI/IP and UDP protocols

4.3 Java

> Java, offıcıally released ın Aprıl 1995, started out as a language desıgned to wrıte portable programs for embedded processors (Edwards, p 6)

> Sun descrıbes Java as follows Java. A sımple, object-orıented, dıstrıbuted, ınterpreted, robust, secure, archıtecture neutral, portable, hıgh-performance, multıthreaded, and dynamıc language (Flanagan, 1997, p 3)

This list describes several features of Java that make it the ideal language for JiniOS Because Java is an interpreted language, the user must use an interpreter running a Java Virtual Machine (JVM) to execute a Java program This means that the Java program will run on any computer that has the interpreter running, and that the Java program itself is architecture neutral As long as a JVM for the operating system exists, the Java program will run on that system Additionally Java dictates the data byte ordering in files, which avoids the common problems encountered when moving binary data from one platform to another Java programs are compiled into platform independent byte-codes that are then run by the JVM During the byte-code generation, the code is verified to ensure that the program is well formed and does not overflow or underflow the stack or contain illegal byte-codes This is only one aspect of the security features of the Java language Another aspect is that all Java programs can be restricted to specific functions to ensure security by specifying a security policy file on the system running the JVM These restrictions are enforced by the SecurityManager class, and all core Java classes that perform sensitive operations ask permission from the SecurityManager before the operation is performed

Another benefit resulting from being an interpreted language is that Java is a dynamic language Java programs are not linked into an executable, as with most other languages such as C++ and Fortran, so classes can be dynamically loaded and instantiated The distributed nature of Java, combined with the dynamic class loading capabilities makes it possible to download and execute classes via the Internet Because Java is a distributed language, it provides support for high-level networking as well as the traditional lower level networking support Distributed applications typically involve starting a new thread of execution in the server to handle communications with each client As a client request is received, a new thread is started to process the client's requests In this way, one server can process all communication with several clients and still listen for new connection requests Multithreading is included in Java

Finally, the Java language is simple and small and has been designed for writing robust and highly reliable programs Software reliability is an extremely important feature when writing a device driver—it must not fail or require any user intervention Exception handling to respond to errors that may occur is essential and is part of the core Java language

Java contains several classes that facilitate serialization (reflection), which involves being able to convert an object into a stream of bytes (marshal) and later reconstruct the object (unmarshal) from this stream Once converted, these bytes can be written to a file or transmitted to another computer via an Ethernet connection Serialization of objects is made much easier by the ability of Java classes to reflect upon themselves Reflection allows Java classes to determine, at run time, the methods, fields and constructors defined for a class Reflection typically allows classes to be serializable merely by extending the Serializable class and without additional programming Recall that devices in JiniOS will transmit their drivers to the requesting clients via a server These drivers will be Java class objects, and it is the serialization aspect of Java that makes this possible

4.4 Remote Method Invocation

Remote method invocation (RMI) is a facility included in the core Java that allows objects to interact with other objects running on remote virtual machines on a network RMI provides a mechanism for obtaining a reference to an object running on a remote host The methods associated with the object can be invoked as if it were running in the same JVM as the client, thereby allowing an application to be distributed across a network RMI handles all the underlying network communication between hosts so applications may be made distributed without having to program the low-level Internet communication.

Because RMI is built into the core Java application programming interface (API), integration of remote objects into an application is almost seamless The first step is to define the interface for the remote object The interface extends the Remote class and only defines the method signatures (method name and argument types) of the methods and the public data items that are to be remotely accessible Next, an implementation of the interface is created which extends the class UnicastRemoteObject and provides the details of the methods in the interface class This is where the methods specified in the interface are actually coded, so all methods specified in the interface must be represented While this file may contain member functions and data other than those outlined in the interface, only those specified in the interface will be remotely accessible The implementation is then processed, by **rmic** for the Sun Java Development

Kernel (JDK), by using the interface and implementation to create a stub and skeleton for the object The skeleton is compiled with client and the stub with the remote object When methods of a distributed object are invoked, they are converted to network requests by the stubs The skeleton reads the request on the host where the object originated and translates them to method calls The results are transmitted back to the client when the method completes These hide the underlying network details so it appears to the client that the method was run locally Figure 4 3 illustrates the functionality of a client/server application in RMI and a sample of an RMI program is located in Appendix A

In order to make the object available it must be exported to a name server, **rmiregistry** for the Sun JDK, by the host and then imported by any client wanting to use it This is accomplished by the rebind and lookup commands The implementation binds an instance of the object to a name in the server (rebind), and the client uses the same name to fetch a copy of the object (lookup) The client uses the interface, containing just the method signatures, to cast the object to an instance of the interface so that the client can then invoke the methods of the object This is because in typical applications the client does not have access to

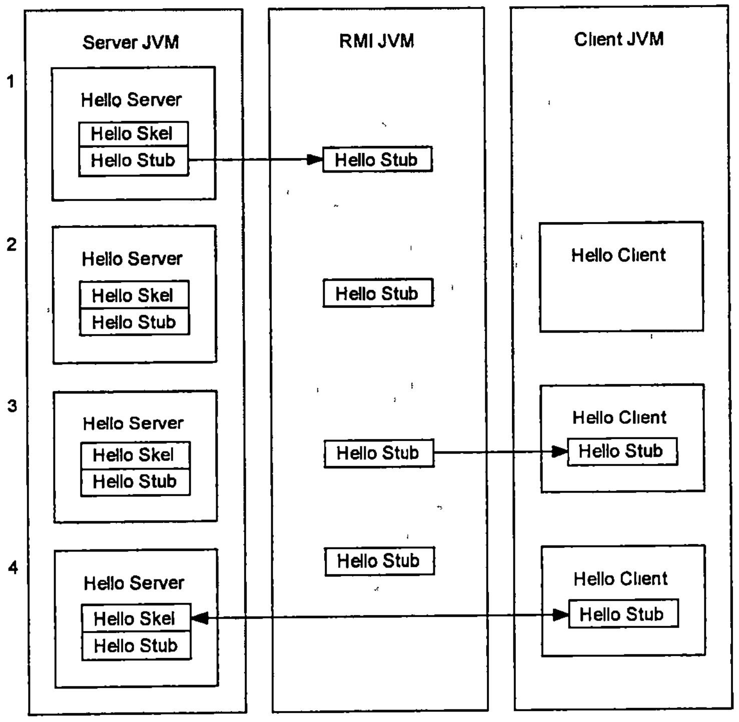


Figure 4 3 RMI client/server model

the implementation, thereby hiding code details from all clients Recall that Java provides techniques for object serialization, which convert an object to a stream of bytes RMI extends this concept but the underlying idea is the same

Having to know beforehand the address of a name server and the name used to register the objects is a major shortcoming when it comes to just using RMI to handle device management In order to use a remote object, the client must first know the IP address or host name of the host running the name server, and the name used to register the object In the case of JiniOS this would mean that each device would be running a copy of **rmiregistry,** which would usually have only one object exported to it Alternatively, each device would have to know the IP address of a name server to use and the clients would have to know the same IP address In both cases we do not know enough information to be able to connect the devices to the client without using predefined IP addresses for either or both The operating system would then have to poll each of the devices for which it somehow got an IP address to determine if it was new, still alive, or no longer present Jini will solve these problems

Jini employs a modification of RMI which solves the addressing and naming problem allowing the device exporter to hide the internal execution details of the driver from the user The user only knows the class interface, not how the class methods are implemented It also lets driver writers focus on the device support software rather than having to worry about the communication details RMI is used for this project to export devices that the operating system does not support sharing, i e , those devices that cannot normally be shared among computers. RMI is used to run an application on a remote system that has a sound card installed from one that has no sound card Sound cards cannot be shared under Windows 95 and, under normal circumstances, cannot be run remotely None of the drivers necessary to communicate with the sound card are available on the local system, so the methods that use them cannot run locally

4.5 Jini

4.5.1 Introduction

Java began as the Oak programming language, part of the Green project, for embedded processors at Sun Microsystems Labs in 1990 Patrick Naughton, Mike Sheridan and James Gosling were "trying to figure out what would be the next wave of computing and how we might catch it We quickly came to the conclusion that at least one of the waves was going to be the convergence of digitally controlled consumer devices and computers" (Gosling) The original vision for Java was to provide a means of supporting groups of consumer-oriented electronic devices that could easily exchange information, but by the time it was re-released as Java in 1995, it had grown into much more While it still provides the ability to securely send and execute code between computers, the original goal of using Java to create groups of easily administered devices was never met

In 1999 Sun Microsystems released the Jini library, which would make the vision of an easily administered distributed network of devices a reality. Many of the same people that worked on the design of the Java programming language were instrumental in the design of the Jini system Among them were Bill Joy, one of the chief inspirations, Jim Waldo, chief architect and designer, Ann Wollrath, inventor of Sun's Remote Method Invocation, and Ken Arnold, creator of Jini's transaction and storage models Jini's lookup and discovery protocols were created by Bob Scheifler, who once led the X Consortium The distributed storage model used for Jini was based on the Linda system from David Gelernter at Yale The Linda programming language was built on the concept of "tuple spaces" and was aimed at distributed versus parallel programming Tuple spaces allow processes to communicate, through space or time, even if they are wholly ignorant of each other Processes communicate and synchronize their activities via a persistent store called a "space" rather than direct communication (Freeman) In fact JavaSpaces, an implementation of the original tuple spaces described in the context of Linda, was later released by Sun Microsystems as a Jini service for storing Java objects

> In some ways the history of Jini is the history of Java—Java is really the fulfillment of the original Java vision of groups of consumer-oriented electronic devices interchanging bit of data and bits of code (Edwards, p 8)

> Jını leverages the Java programmıng language to provıde a sımple substrate for dıstrıbuted programmıng Jını moves us from a world ın whıch the "system" ıs the ındıvıdual networked devıce, to one ın whıch the "system" ıs the collectıon of all these devıces workıng together (Edwards, p xxıx)

Jını extends the Java core to provıde the mechanısms necessary to handle a network of embedded processors communıcatıng to create one complete system It buılds on the networkıng capabılıtıes, RMI and other propertıes ınherent ın Java to create a lıbrary of Java classes taılored for communıcatıon between remote devıces and provıdes the tools to set up servers that manage sets of clıent devıces "Jını allows anythıng wıth a processor, some memory, and a network connectıon to offer servıces to other entıtıes on the network or to use the servıces that are so offered" (Waldo, 1994)

Jını facılıtates the creatıon of a dıstrıbuted system composed of remote devıces In the world of dıstrıbuted systems, the dıstınctıon between server and clıent becomes clouded Servers can also be clıents and clıents servers So for the purpose of thıs research,

- server wıll ındıcate the Jını server,
- clıent wıll be the applıcatıon downloadıng from the server,
- drıver or proxy wıll be the applıcatıon loaded ın the server, and
- devıce wıll be the hardware component and applıcatıon loadıng the drıver to the server

4.5.2 Jını Server

A Jını server acts as a reposıtory for methods and data Recall that one of the major shortcomıngs of usıng RMI ıs that the applıcatıon wantıng to access the remote methods had to know the IP address of the host on whıch the RMI name server ıs runnıng The Jını server uses Internet multıcast to solve thıs problem When a server ıs created, ıt announces ıts presence to all hosts on the same subnet, ı e , all hosts wıth the same first three numbers ın theır IP address For example, gıven a Jını server wıth the IP address 192 168 2 13, all hosts wıth an IP address between 192 168 2 1 and 192 168 2 255, ınclusıve, would receıve a request All hosts wantıng to partıcıpate wıth thıs server respond and waıt further communıcatıon from a clıent The server stores the IP addresses of each partıcıpant for use later In thıs way, the server merely acts as a "go-between". That ıs, after a devıce has uploaded ıts drıver to a server and a clıent has downloaded the drıver, the clıent communıcates dırectly wıth the devıce usıng the methods provıded by the drıver In addıtıon, since a server is responsible for providing the initial communication between the client and a device, and provides nothing after communication between them has been established, devices may be registered with several servers

The ability to move code from the service to the client is what distinguishes Jini from CORBA and DCOM In those systems, the code used to communicate with a service is associated with the client and knows how to transfer information to the service via a static protocol defined in terms of an interface definition language (IDL) The code being moved is called a proxy and is central to the Jini system A proxy is a local object that stands for a remote object so that the combination of the proxy and the service form a single object that is itself distributed (Waldo, 1994)

Both clients and devices also use multicasting to identify all the Jini servers on the subnet All Jini servers that receive a request for identification respond to the requestor The Jini software stores the IP address of the server for use later, which hides the need to know IP addresses or host names as long as all servers, clients and devices are on the same subnet This is usually the case in a home or even office environment Jini makes provisions for servers that reside outside the subnet, but in this case the host name or IP address must be used to establish initial communication between the device and the server and between the client and the server As before, the IP address of the server is required only to store and retrieve drivers but not between the clients and the devices In this way, devices such as printers can register themselves with several known small networks to be visible to all clients Conversely, each smaller network might consult known servers on other networks that contain devices that clients might want to use. In fact, Jini allows servers to register with other servers, thereby making themselves visible outside their normal range

Another important responsibility of the Jini server is to guarantee that all devices associated with a server are still valid Communication between the server and a device is via UDP to avoid the overhead of maintaining a permanent connection Unfortunately, this means that neither computer in a UDP connection knows when the other is no longer accessible because of either line or system failure If the server did not communicate periodically with a device to verify that it is still active, then a client would be able to download drivers for devices that are no longer valid While the client could recover from this when it tries to access the device, it gives an incorrect snapshot of the system Verification allows the server to clean up limited resources by removing drivers for devices that are no longer accessible

4.5.3 Leasing

Jini uses leasing to determine if a device is still available The server sends a request to every device registered with it at regular intervals and waits for a response from each one The duration, known as the lease period, is specified by the device when it registers with the server If the device fails to respond to this hail, it is removed from the server and all clients using the driver are notified that the device is no longer available If the device comes up later, it will re-install its driver and all clients may then download them again This is known as a self-healing network—the loss of a device is not catastrophic and is recoverable without user intervention The device also uses leasing to ensure the server with whom it is registered is still available Aside from specifying the lease duration, the Jini software hides the leasing mechanisms from the devices and clients and handles all registering and renewal of leases automatically Once a device has identified a Jini server, the device uploads its driver (Figure 4 4) Similarly, a client that has identified a server, downloads its driver when necessary (Figure 4 5)

4.5.4 Programs

When a Jini program is executed, it identifies available Jini servers and the services available from each If the program is associated with a device, it uploads its driver to the server If the program is a client that wishes to use one of the services, it contacts the server and requests the object associated with the desired service The server then transfers the object (driver) to the client and the client then contacts the device publishing the service using the methods of the driver If the object represents a remote object, these method calls will be translated into network requests and are run on the device, and results are then communicated using RMI If not, they are run on the client and communication to the device must be established another way (usually via a private protocol running on TCP or UDP) Once the initial communication has been established, the requirement for Java is no longer in effect The device may choose to use any programming language to service the client, as long as the communication satisfies the

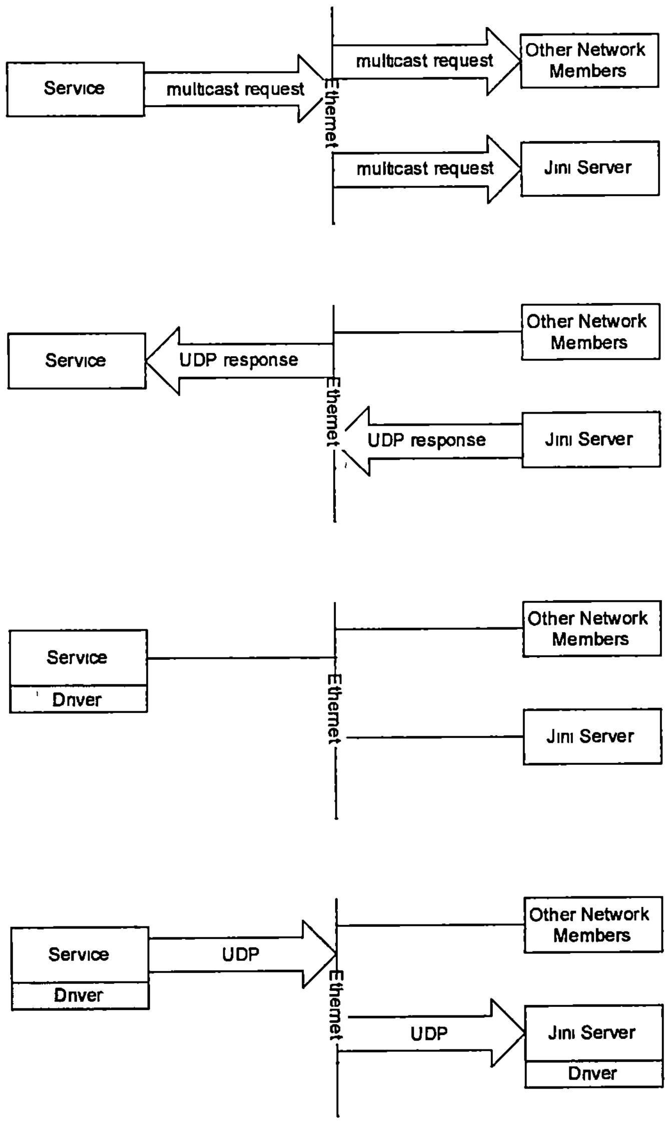


Figure 4 4 Discovery / Join

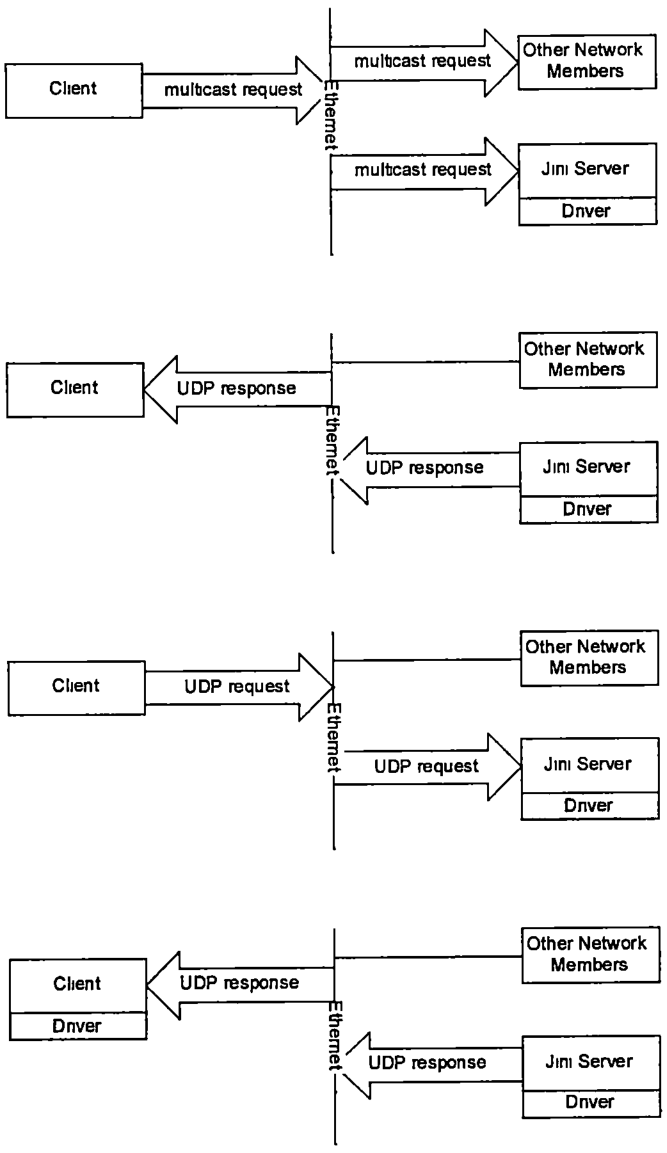


Figure 4 5 Discovery / Lookup

established protocol This may prove more difficult on the client side because the client is running methods of the driver to communicate and the driver is written in Java However, the driver may be written in such a way as to provide additional language choices to client

Regardless of how the communication is handled, the client must know the method names and their arguments before they can be used Recall that when using remote method invocation (RMI) a shared program comes in two parts the interface and the implementation Jini too follows this convention to publish the structure of shared objects This means that services grouped by type must follow specific, advertised interface guidelines in order to be used by a client, regardless of how the methods are eventually implemented For example, all printer devices must provide drivers with the same core method signatures They may provide drivers with additional methods unique to their printers but the core method signatures must be identical Jini does not specify or enforce these standard device interfaces—they will have to be determined by the various device manufacturers

4.5.5 Jini details

Jini is constructed of layers on top of Java the same way as the Java Foundation Classes (JFC) and the Java Database Connectivity (JDBC) libraries (Figure 4 6) but at its core, Jini is pure Java

> On top is a directory service, based on a "lookup" mechanism that allows different Jini-enabled devices and applications to register and be seen on the network. The next-level service is persistence, provided by JavaSpaces technology, which stores objects so that other users or applications can retrieve them Below that, a set of protocols based on Java's Remote Method Invocation enables objects to communicate and pass each other code And finally a boot, join, and discover protocol allows Jini-compatible devices, users, applications to announce themselves to the network and register in a discovery (Kelly, p 132)

Jini is built around five essential concepts several of which are familiar from the discussion on distributed systems discovery, lookup, leasing, remote events, and transactions Discovery is the mechanism described above, used by devices to find communities of servers on the network and uploading (joining) their drivers to them Lookup is used by clients to find servers Leasing, described previously, is one of the most important concepts of Jini because it ensures that a community will recover from the loss of any its

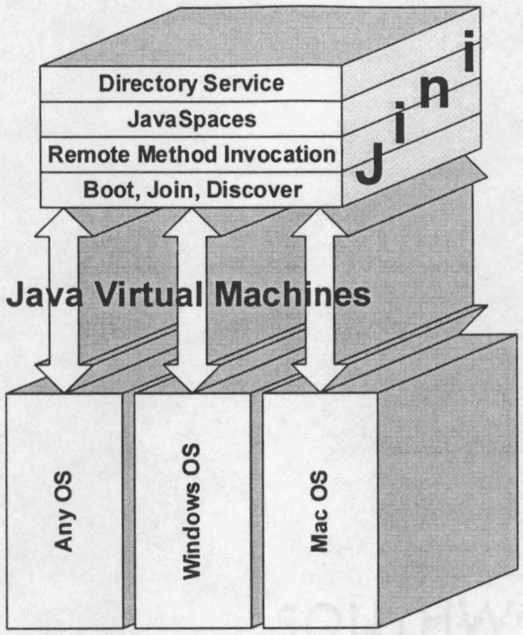


Figure 4.6. Jini structure.

members. Leasing is also used by several JiniOS devices to ensure that the device is made available again when the client terminates. Remote events allow members to notify each other of changes in their state without each member having to poll the rest for changes. Jini makes extensive use of events because of the non-predictable nature in dealing with network applications. For instance, when a request is made for servers, rather that waiting a specified time for responses, Jini establishes an event to monitor and returns control back to the calling routine, and then agrees to invoke the specified callback when the event is triggered. Because there is no way of knowing beforehand how many servers will respond or how long it will take them to respond, we cannot simply wait. The last concept, transaction, is implemented in Jini to provide two-phase commit transactions for those operations that may require them. This ensures that those services that require an all-or-nothing transaction process have a Java solution available to them. The transaction feature of Jini was not used for JiniOS but was simulated at a low level by some devices, a hard drive for example, by requiring an acknowledgement after every network-related operation, essentially committing every operation.

When a new service is created or an existing service is modified a unique id, called the service id, is associated with it The same service id should be used every time the service is registered and should be the same for all Jini servers with which it is registered A client can decide if a service found on several servers is just the same service registered everywhere or different services by comparing the service ids Additionally if a service fails and then reappears, the client can determine if the driver associated with the service has changed and behave accordingly If the driver changes, a new service id should be created so that all clients know that their drivers are no longer valid and must be replaced Finally, the service id can be used to distinguish between drivers of the same type that may implement vender-specific methods For example, drivers supplied Hewlett-Packard and Epson are both printer drivers and must implement the standard interface, but they may include additional methods that take advantage of their hardware The service id could be used to distinguish between them The technique used to generate the service id is located in Appendix B

> Jini service Ids are 128 bits long Even if the world's population reaches 100 billion, and each of these people have, say, a million Jini-enabled VCR's, cameras, and DVD players to their name assigning unique Ids to these devices would only take around 64 bits of the service ID address space Jini uses a somewhat more sophisticated technique than just cranking out random 128-bit numbers, though But basically the service id is created by stuffing together 60 bits of system clock, expressed in 100 nanosecond chunks since the year 1582, a bunch of random "noise" thrown in for good measure, and, on some implementations, a unique host address for the lookup service (usually its ethernet address, if it has one) This scheme guarantees unique numbers until the year 3400 AD, at which point the clock rolls over Even then, however, the host identifiers and random number components will likely provide uniqueness, although it can no longer be guaranteed (Edwards, pp 283-284)

In addition to a service id, each service has an Entry object associated with it The Entry contains information about a service such as the product name, manufacturer, vendor, version, model, and serial number that a client uses to decide if this is the object it needs The Entry object corresponds to the expressions used in the Intentional Naming Systems (INS) to classify and select services

The services that Jini requires are

- A simple Web server Jini requires this facility because when downloaded code is needed through RMI, the actual transmission of the code happens via the HTTP protocol . A common configuration is to run an HTTP server on each host that needs to provide downloadable code to other applications

- The RMI activation daemon allows objects that mat be invoked rarely to essentially "go to sleep," and be automatically awakened when they are needed The RMI activation daemon manages the transition between active and inactive states for these objects, and is used extensively by the other core Jini run-time services At a minimum, you will need to run the activation daemon on each host that runs a lookup service, described below

- A lookup service keeps track of the currently active Jini services that are available on a LAN (Edwards, p 23)

JiniOS, discussed in greater detail in the next section, is configured similar to that described above (Figures 4 7 and 4 8) There exists a personal computer running a lookup service, a RMI daemon and an HTTP server pointing to the Jini code This same computer runs all clients and has another HTTP server pointing to a client download area to handle RMI requests for objects Each device runs an HTTP server pointing to the area where the driver to be transferred is located All leases are set to expire after ten minutes so the entire system will be stable and correct ten minutes after a device or server or client terminates Most devices contain Close methods that allow a client to disconnect from a device when finished so leasing is employed primarily to recover from software or hardware failures or intentional system configuration changes

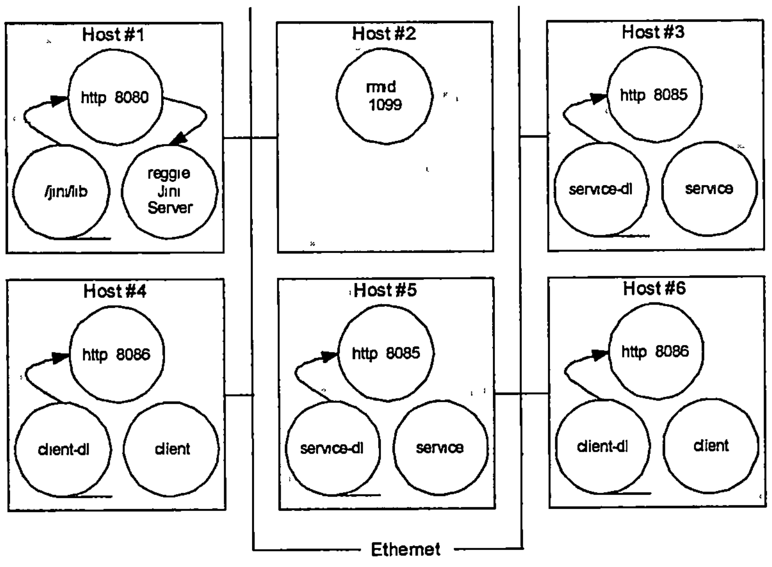


Figure 4 7 Configuration required by Jini

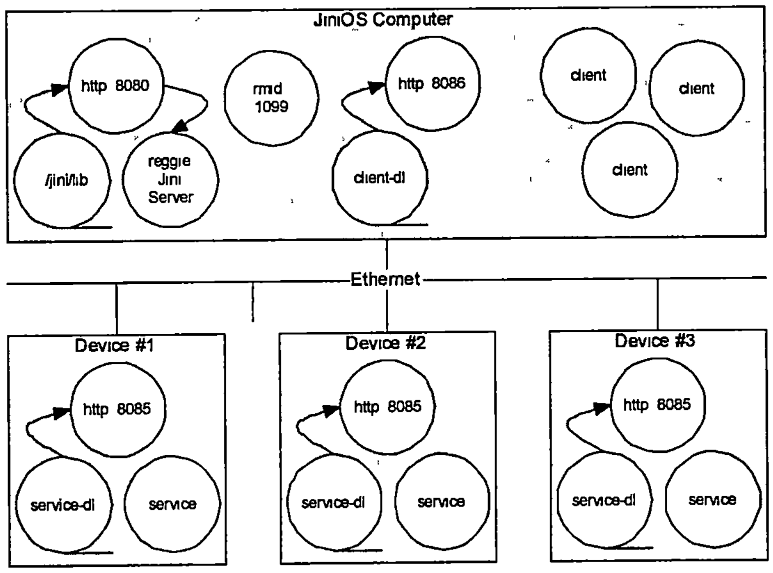


Figure 4 8 Configuration used by JiniOS

Chapter 5
Implementation of JiniOS

5.1 Project Description

The goal of this project is to create a simulation of a computer comprised of a federation of connected devices I simulated several devices by writing their device drivers and a client that accesses and uses these devices The devices must be able to automatically install themselves by uploading their driver software to the Jini server and must be able to tolerate the loss of the Jini server and any client system failure The client must be able to tolerate the loss of devices and be able to accept the addition of new devices without requiring the system to be rebooted or any user intervention Both must tolerate the loss of connection to the server and each other Success will be determined by how well the devices, servers, and clients behave given these circumstances

Realizing that there are no devices currently on the market that meet all the hardware requirements previously outlined (CPU, memory, Ethernet connection, etc), I configured four personal computers with the necessary devices attached to provide the hardware components (Figures 5 1 and 5 2) to create the appearance of a single, complete computer system (Figure 5 3) I selected a keyboard, monitor (text only), hard drive, sound card and printer as the devices to model as JiniOS devices For example, the keyboard is a complete personal computer with a keyboard attached. Likewise, the monitor is another complete personal computer with a monitor attached The same holds true for the sound card and hard drive In order to populate the server with enough device choices, several systems represent more than one device The computer that exports the keyboard, for example, also exports a hard drive Similarly, the monitor and sound card computer systems export hard drives However, each configuration behaves independently from the devices that reside on the same physical computer with it The keyboard, for example, does not use the fact that its PC host also exports a disk drive and visa versa

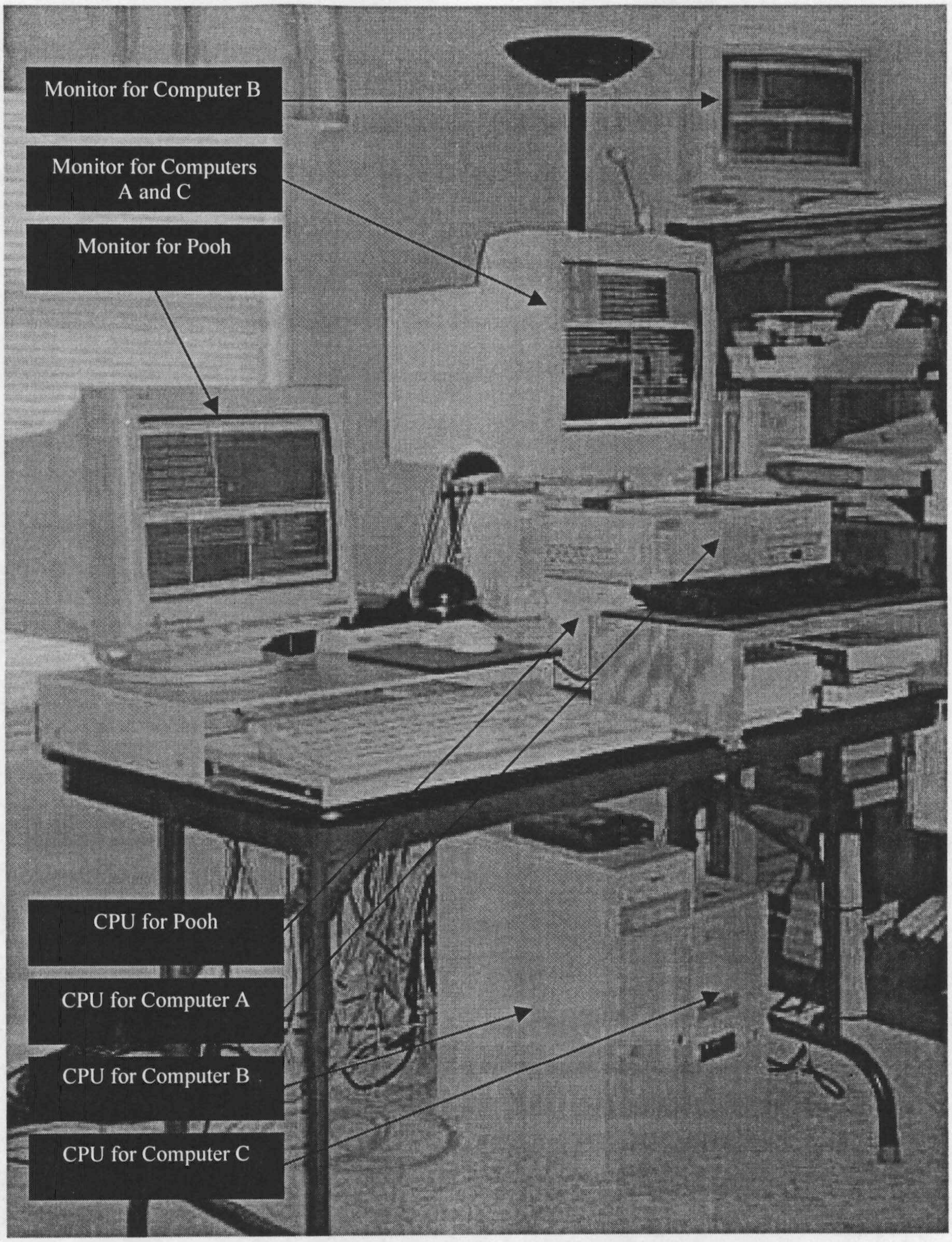


Figure 5.1. System photograph.

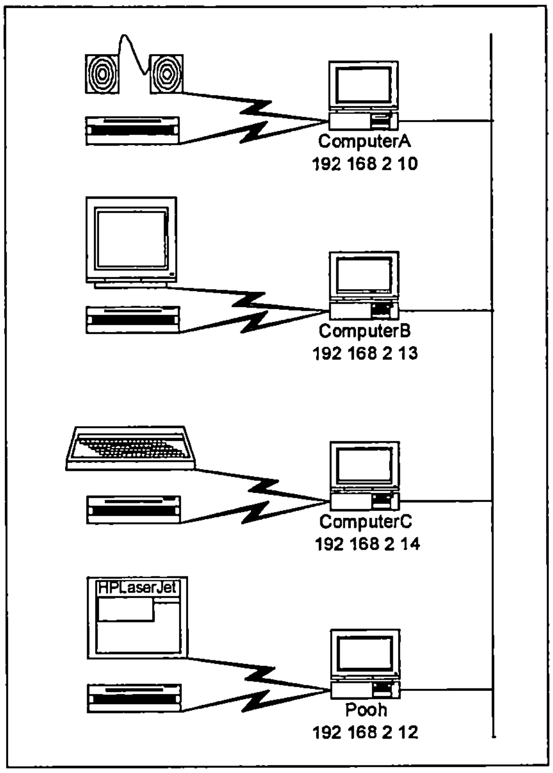


Figure 5 2 Actual system

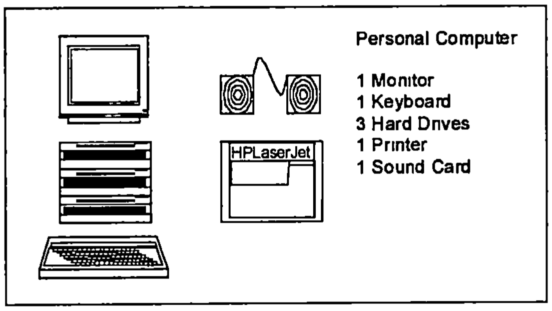


Figure 5 3 Virtual system

All four computers are connected to each other via Ethernet and all are on the same subnet (Figure 5.4). The IP addresses “192.168.2.xx” were selected because this subnet is typically reserved for independent home networks thus guaranteeing there will be no address conflicts between this local network and the World Wide Web. In an complete JiniOS system the IP addresses of the devices can not be fixed since there is no way for the manufacturer to know beforehand the local subnet (“192.168.2” in the above address). Recall that Jini uses multicasting on the local network to identify servers and must know this to perform the multicast. One possible solution would be to assign an IP address to a device during installation but this would defeat the whole automatic-installation-only-when-you-need-it purpose of JiniOS. A better solution is to use a technique such as Dynamic Host Configuration Protocol (DHCP) which permits automated allocation of IP addresses. A DHCP server monitors the multicast address DHCP-AGENTS.MCAST.NET (224.0.0.0) ports 67 and 68 and assigns an IP address to a client as it is requested (Harold). Each address is taken from a pool of addresses and is returned to the pool when the client is finished. DHCP uses leasing to determine when a client is no longer available; i.e., when the client no longer renews the lease, the address is returned to the pool (Hunt). DHCP was not used for this project but it or some other mechanism for automatically assigning IP addresses without a network administrator would be required in a final JiniOS system. DHCP may not be necessary in the future because the new version of the IP header, IPV6, allows for autoconfiguration. This lets computers assign themselves a permanent address based on the manufacturer of the network card and its 48-bit Ethernet address (Montfort).

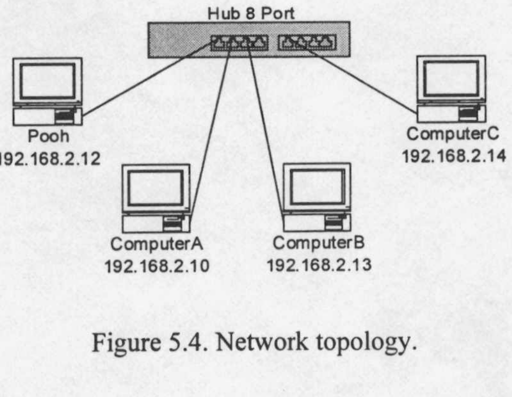


Figure 5.4. Network topology.

All computers are running Windows 95, but this is not a requirement for JiniOS Windows 95 was selected because Jini requires JDK 1 2 1 or higher and was only available for Windows 95 that the time the project was started Also the universal portability of Java programs across different platforms and operating systems has already been established and does not have to be proven by this project Since Jini is written in Java and all the services and clients are written in Java and all communication is based in standard Ethernet protocols, the fact that all the systems are Windows 95 does not affect the conclusions

The keyboard, monitor and hard disk devices use a tailored protocol to communicate with their clients They use the Jini server to distribute their respective drivers but methods within these drivers establish a dedicated TCP connection between the device and client Each employs leasing techniques to monitor the health of the connection The sound card device employs RMI and a dedicated HTTP server This is because the program to play the sound must be run on the computer with the sound card installed An HTTP server is used to transfer the music as streaming audio rather than transferring the entire sound file to the device before playing it By using streaming audio, the remote device does not have to store the file locally and it can begin playing almost immediately Likewise, the printer requires a print server running on the computer to which the printer is connected By using RMI, the client appears to running a print server locally when, in fact, the file to be printed is transmitted to a remote print server The software that supports the hard drive, the sound card and printer devices is written in Java The software that supports the keyboard and the monitor devices is written in the C^{++} programming language C^{++} was chosen for the keyboard because immediate access to the key pressed was required C^{++} was used for the monitor because the device software for the monitor was similar to that for the keyboard

Sample source code for the keyboard device and a simple client that accesses it is located in Appendix C The audio player and printer devices are somewhat more complicated by the fact that the driver that the client downloads must be run using RMI The source code for these devices is not included in this document but source code, along with a copy of this thesis, will be stored at the following Internet address http //www utsi edu/cs/jinios and may be viewed at any time

5.2 Device Software

The implementation of each device can be broken down into three basic parts the interface, the driver and the device software The interface contains the class definition that the client will use to create a usable object from the one that is transferred from the Jini server when the driver is downloaded It is the only part of the driver for which the client has to have the source code since it contains the signatures of the methods that the client will invoke to use the driver The object that is transferred from the Jini Server is of the type Object, the super class of all Java objects The client casts this object to an object of the type of the interface thereby gaining access to the methods and data. It is important to note that Java enforces this cast, i e , Java will not allow an invalid cast if the original object is not either of the interface type or one of its descendants Figure 5 5 illustrates the type of information generally found in the interface

This interface guarantees to the client that the driver has implemented three methods open, close and read None of the three take an argument and only read returns a value After the client has downloaded the driver object from the Jini server, the object will be cast to the KeyboardServiceInterface The client will then invoke the open method to establish a connection to the device, the read method to read characters from the keyboard, and finally the close method when the client is finished with this keyboard device

The driver implements the interface for the device While the driver may contain additional methods necessary to implement the task, it must define those methods listed in the interface and only those methods and data specified in the interface are accessible to the client The driver object is instantiated on the device but runs on the client's machine This object will be transferred to each client and will provide the only connection between the client and the device In order to accomplish this, it also must implement the Serializable class so that it can be serialized and transferred to the Jini server and then deserialize by the

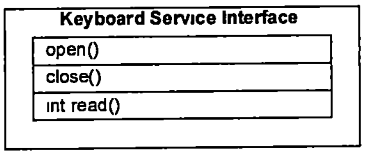


Figure 5 5 Sample keyboard service interface client when it is downloaded Typically making a class serializable involves nothing more than including the Serializable class in the implements list of the class definition This ability of Jini to transfer the program necessary to implement the driver to the client only when the client requests it is what distinguishes JiniOS from other device driver mechanisms

This driver, Figure 5 6, implements the methods dictated in the interface – namely open, close and read KeyboardService uses a private TCP connection, as opposed to using RMI, to communicate between the client and the device The driver is created on the device, uploaded to the Jini server and downloaded to the client The client does not create a new instance of KeyboardService, but instead casts the object downloaded from the Jini server as a KeyboardServiceInterface object. This distinction is important because at this point the client does not know the IP address of the device and cannot initiate communication with the device without it nor does the device know the address of the client When a KeyboardService object is created (on the device), the host name and port number of the device are supplied to the constructor They are then serialized and stored on the Jini server when the driver is registered The client downloads the driver and casts it to KeyboardServiceInterface, and the hostname and

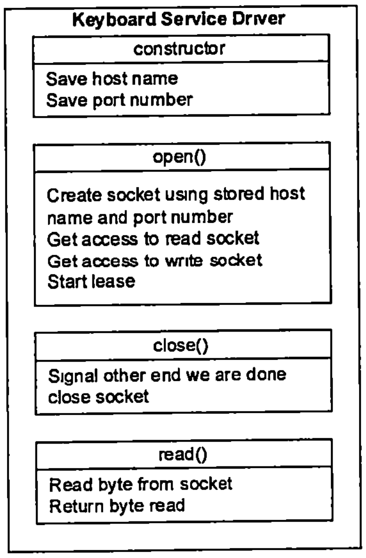


Figure 5 6 Sample keyboard service driver port number that were already set are used by class methods to establish a connection back to the device The device is listening to the specific port waiting for a client connection Since the client supplies its IP address and port number in the connection protocol, both the device and the client now have a means of communication This allows the client to initiate communicate with a device without first knowing its address or the port number of the device

The open, close and read method details are straightforward Open creates a TCP socket connection to the device and assigns the class variables put and get to the output and input socket variables, respectively The read method uses the readByte method of the get variable to read a single byte from the socket This translates to reading a byte from the device that the socket is connected, the keyboard in this case This read blocks until a key is received The close method terminates the connection between the client and the device

Recall that Jini uses leasing throughout to monitor the health of services and maintain the overall integrity of the network The KeyboardService also uses leasing to monitor the health of the connection between the device and the client The keyboard may remain unused for extensive periods while the user employs other input mechanisms such as a mouse or barcode scanner or the computer sits idle In addition, this particular device is designed to communicate with only one client at a time While other devices, such as hard drives, may allow several clients to connect simultaneously, we would expect a keyboard to be dedicated to a single client Even though the KeyboardService driver provides a method to terminate communication, we must have some mechanism to determine when a client is no longer active in order to free the keyboard for another client We must be prepared in the event the client does not invoke the close method either because of poor programming practices or system or network failure The solution is to use leasing - at the same time a client connects to the device via the open method a lease is automatically set up and maintained in a separate thread Every time the lease runs out, the client lease thread renews it The same lease period is monitored on the device If the lease is not renewed the device assumes that the client has terminated and the connection is no longer necessary whereupon the connection is closed and the device returns to waiting for a connection

Next we have the program that is responsible for locating a Jini server and registering the driver with it, and, if additional programs or threads are necessary to service the client, it is responsible for starting them This same program is responsible for maintaining the lease contract with the Jini server after the driver has been registered Remember that Jini uses leasing extensively to provide the self-healing property of Jini federations As long as the lease is renewed the Jini server will keep and distribute a copy of the driver, however, if the lease expires, the server will drop it and notify all clients that have copies of it

This program will typically be the only class on the device side that is Jini, that is, this is the only class that includes any Jini-specific code Most services can be broken down into several simple tasks persistence, discovery, instantiation and registration and of these only discovery and registration are specific to Jini These involve locating a Jini server and registering and maintaining leases for drivers Jini provides several classes to support the core Jini mechanisms as well as several convenience classes to make the task easier For example the JoinManager class takes care of participating in discovery, propagating attributes, handling groups, and maintaining leases Using these convenience classes means that most devices will follow a standard boilerplate, illustrated in Figure 5 7, to register with a Jini server as long as they are willing to allow the convenience class to take care of everything

The constructor first ensures that a security manager is running - required to run any remote applications Then the current hostname and a port number are used to create an instance of the KeyboardService object that will later be sent to the Jini server As stated before, the object is created on the device but is run on the client Next we follow one of two basic paths either no service id has been assigned yet or we need to reuse an existing service id Which path to take is decided in the constructor by whether or not the serialization file exists or not If this file does not exist then no service id has been assigned and the method register will be invoked, otherwise, the method reregister will be invoked to use the existing service id stored in the file

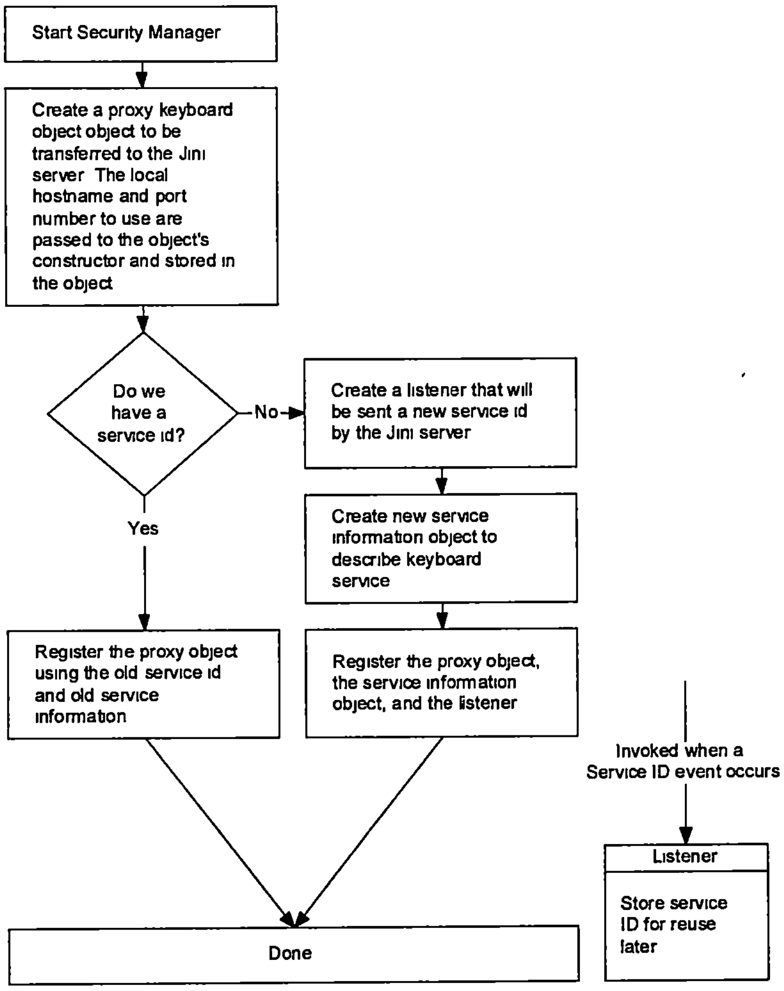

Start Security Manager
Create a proxy keyboard object object to be transferred to the Jini server The local hostname and port number to use are passed to the object's constructor and stored in the object
Do we have a service id?
No
Yes
Create a listener that will be sent a new service id by the Jini server
Create new service information object to describe keyboard service
Register the proxy object using the old service id and old service information
Register the proxy object, the service information object, and the listener
Invoked when a Service ID event occurs
Listener
Store service ID for reuse later
Done

Figure 5 7 Service flowchart

All services stored in a Jini server must have a service id which may be a previously assigned one or, in the case of a new or modified service, a new one Rather than make the user contact a Jini server for an id and wait for that id and then us it to register a service the JoinManager class has two forms One to register pre-existing service and one to register new ones In the case of the latter, the user sets up a class one of whose methods will be invoked when a service id is created In this way, the registering program can proceed in parallel while the service is being registered Once a JoinManager has been created, transfer of control is return to the main routine which will start addition threads to handle client requests or just wait in an infinite sleep The process of registering the proxy object with responding Jini servers is event-driven and occurs in parallel with this loop As each server identifies itself to the JoinManager, the object is automatically registered with it by the Join Manager

The last item to cover is the program or thread that runs on the device and services the requests from the clients It functions much like the BIOS and device driver in the current operating system in that it translates high-level client requests into low-level, device-specific commands In the case of the keyboard, this is a separate program that reads from the keyboard and transfers the data to the client In other cases, the hard drive and audio player, it is another thread of execution and RMI, respectively, in the devices software discussed above In all cases since it runs on the device, it should be started by the same process that starts the device software during the device boot The keyboard device was written as a separate program because we needed access to the key pressed without requiring the user to press the Enter or Return key Typically keyboard input is buffered until the Return key is pressed, but we are able to circumvent this process using a C^{++} program to collect pressed keys As stated before not all programs in JiniOS have to be Java as is shown by this example This program is straightforward but heavily dependent on the Windows operating system It creates three threads, LeaseThread, SignalThread and ClientThread which are responsible for listening to lease renewal requests, close requests and sending the pressed key to the client, respectively It allows only one connection at a time and all three threads must be terminated before the next request for ownership is serviced. If any one thread terminates the system automatically terminates the remaining two

One interesting thing to note is that data transferred from a C or C++ program to a Java program via Ethernet may encounter some word size and byte order problems Java is very strict about both of these issues, however, most other programming languages are not, so some consideration must be given to formatting the data to a format that Java will understand

5.3 Client Software

A client program in JiniOS has to locate a Jini server, identify the drivers available on it, download the drivers for the devices it needs, and access the devices via the methods provided in the driver (Figure 5 8) As with the device software, first the client starts a security manager to ensure access to the device and then broadcasts a request that all Jini servers identify themselves Unlike the device software, however, the client has to set up an object to monitor replies Recall that in the device software almost everything was handled by a convenience class (JoinManager), but in this case the client needs to know the servers that respond so that it can query them for installed devices Since there is no way of knowing beforehand how many servers will respond or even how long it will take to get a response, the discovery process is event-driven This is similar to the way service id's were handled in the device software The client provides a function that the Jini software will use to notify the client every time a Jini server responds That function then uses a template to identify those drivers that the client is interested in and stores them for later use

As was the case for service id's, the discovery process is handled by a separate thread that runs in parallel to the main program so that the program is not blocked until all the Jini servers have responded This allows the main program to continue without having to wait for all the devices to finish loading Of course there may be some cases where the main needs to wait until a particular device is available, the first time a new printer is selected for example, so a mechanism must be built into the client software to wait for a specific device Additionally, when Jini performs the discovery process it arranges a lease with all the servers that respond so that the client is notified whenever the status of any of its registered drivers are modified This allows the clients to reload any changed drivers to add new ones if they represent devices of interest Lastly, recall that as Jini servers are created they broadcast their presence Since the discovery process runs as a separate thread in the main program, the process of examining new servers for devices of

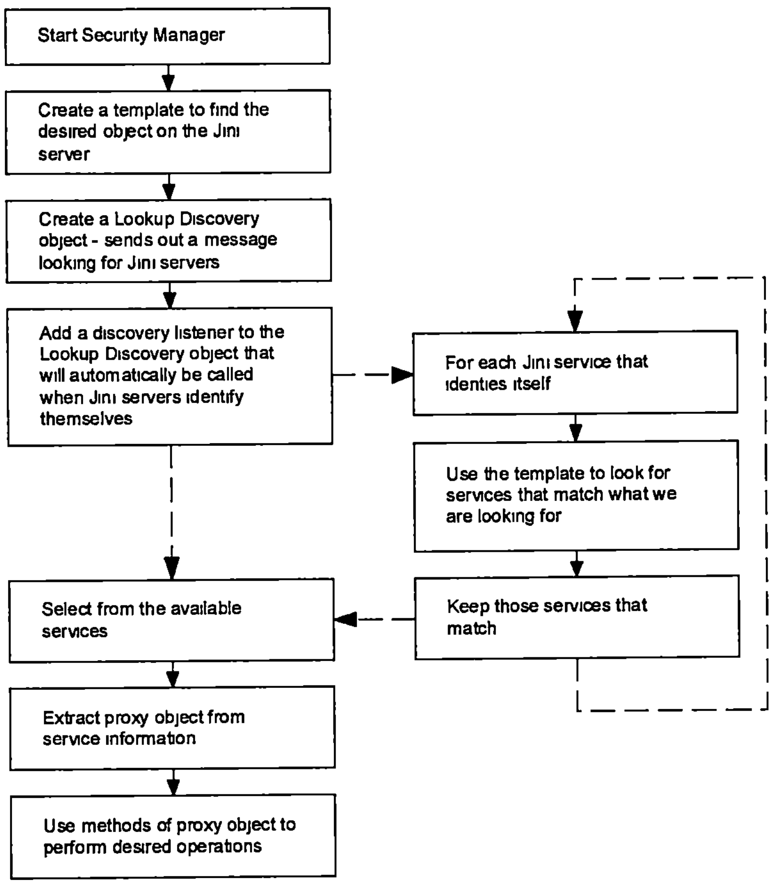

Start Security Manager
Create a template to find the desired object on the Jini server
Create a Lookup Discovery object - sends out a message looking for Jini servers
Add a discovery listener to the Lookup Discovery object that will automatically be called when Jini servers identify themselves
For each Jini service that identies itself
Use the template to look for services that match what we are looking for
Keep those services that match
Select from the available services
Extract proxy object from service information
Use methods of proxy object to perform desired operations

Figure 5 8 Client flowchart

interest runs in the background So whenever a new device is needed by a client the list of available devices is scanned and the one that most closely matches is used

Figure 5 9 illustrates the complete process for a single device The keyboard device uploads its driver to a Jini server and the client downloads that driver from the same server The client then uses the driver to communicate directly with the keyboard device Figure 5 10 illustrates the same process for all the devices in this project

Ideally each client program should be insulated by the operating system from having to perform the discovery process just to access a device (Figure 5 11) Much as the BIOS does today, the operating system should maintain a list of available common devices and allow access to them via a high-level interface This list would be derived information gleaned from local Jini servers, and it would be automatically updated as the network topology changes Less commonly used devices could be actively searched for and attached by the user thereby avoiding cluttering the interface with little-used devices Since one of the goals of this project was to demonstrate the feasibility of Jini networked devices, I decided not to go to this next step but instead let the client manage its own devices as illustrated in Figure 5 10

5.4 Starting the System

Each client may run a Jini server but this is not necessary as long as there is at least one running on each subnet or isolated network Jini does not place any limits on the number of servers running on a network nor on how many different servers a particular service can be registered For the purpose of this project, JiniOS runs a Jini server on each complete personal computer In a production environment, however, we would not expect a server on each desktop but rather one or two per office or even per floor or, in the case of a home system, one or two per home

The process of running keyboard example is broken into three parts code necessary for the Jini server, code necessary for the keyboard driver and code necessary for the keyboard client The program necessary to run the keyboard example is located in Appendix E

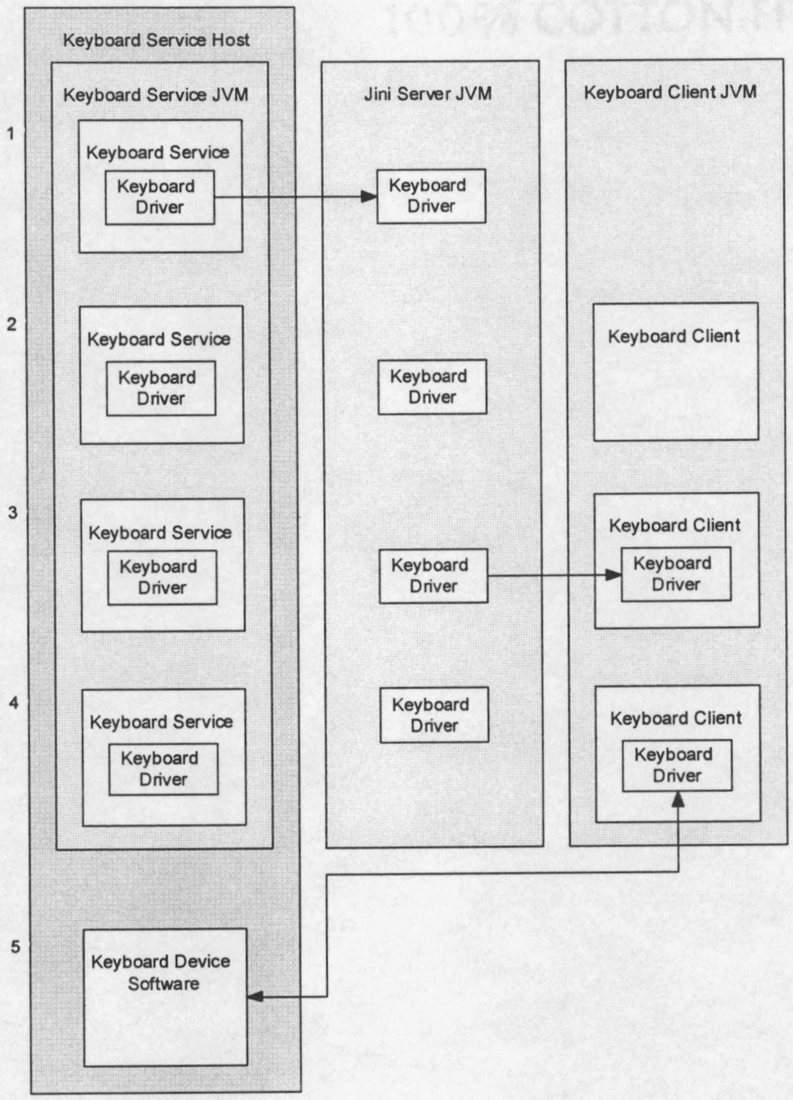


Figure 5.9. Keyboard service and client.

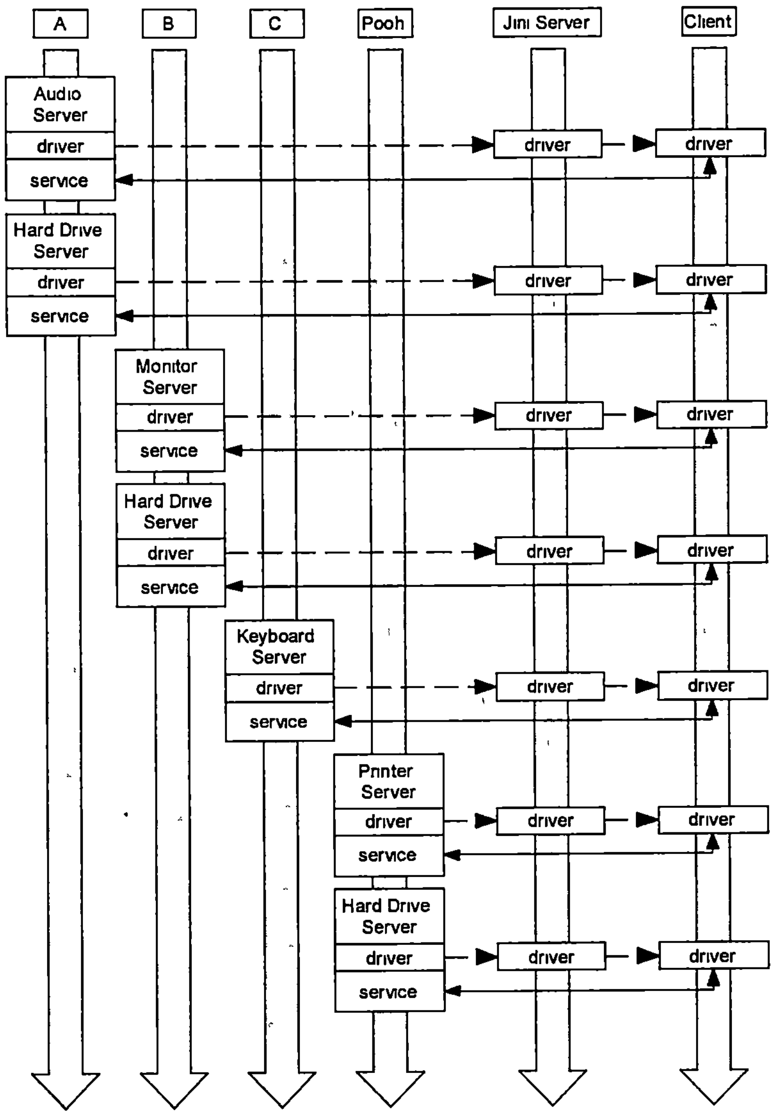


Figure 5 10 Complete project

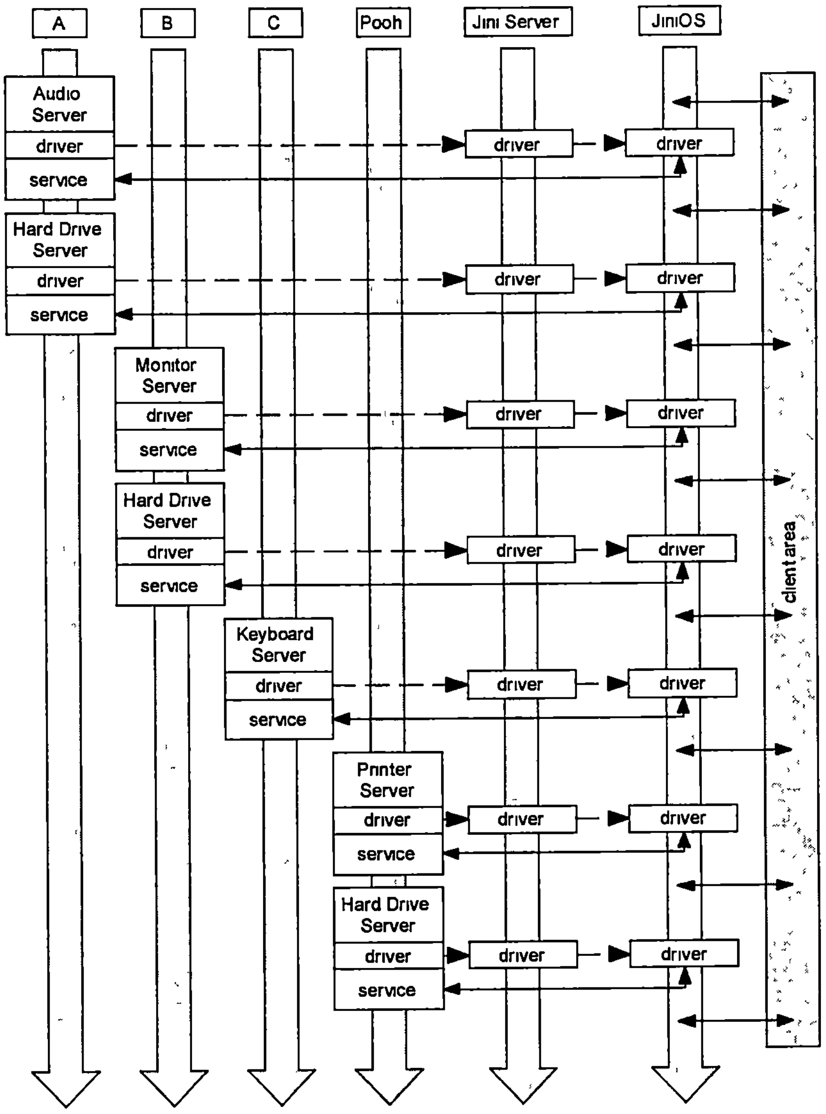


Figure 5 11. Ideal JiniOS

Chapter 6
Evaluation and Conclusions

6.1 Evaluation

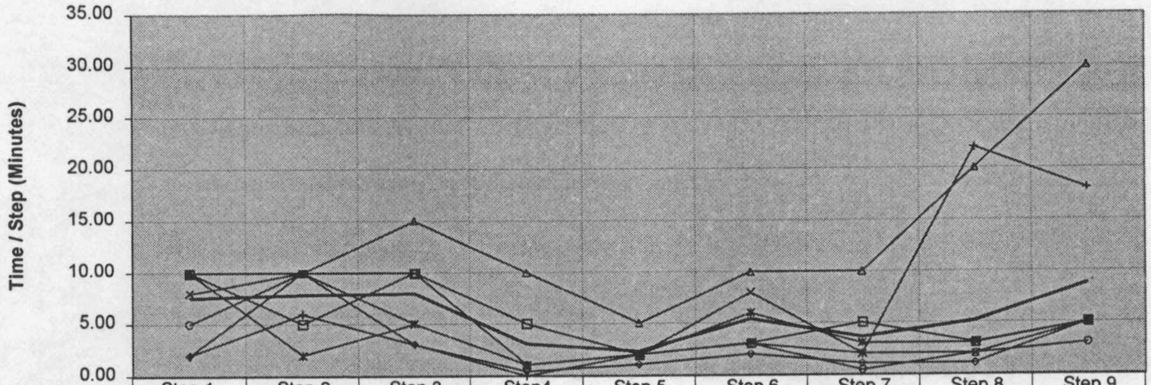
In addition to writing device drivers for the keyboard, monitor, hard drive, printer and sound card I wrote a single client (JiniPC) that accesses all of them I had written individual clients to validate each device separately but the intent of this last program was to validate a complete system composed of networked devices and to verify the 'self-healing' nature of the network All clients and JiniPC performed as expected The JiniPC program was able to access all the devices as they became visible without any user intervention Hardware failure was simulated by stopping the service and repair was simulated by restarting it In all cases the JiniPC program was able to recover once the device reappeared Based on these results, this part of the project was a complete success

The primary goal of JiniOS was to make the process of installing and maintaining the devices easier. I wanted be able to replace the antiquated methods used for installing new devices with one that allowed new devices to install themselves

The installation of a hard drive is probably the most difficult task a typical user would attempt I conducted an informal survey (Appendix F) to determine the minimum time that it takes to install a second hard drive in an existing system under ideal circumstances (enough room in the case, enough connectors in the ribbon, enough power connectors, etc) The survey form was sent to people with various backgrounds but all had some experience installing a hard drive One individual was timed performing the installation of the hard drive to validate the estimated times The results of the survey are located in Tables 6 1, 6 2, and 6 3

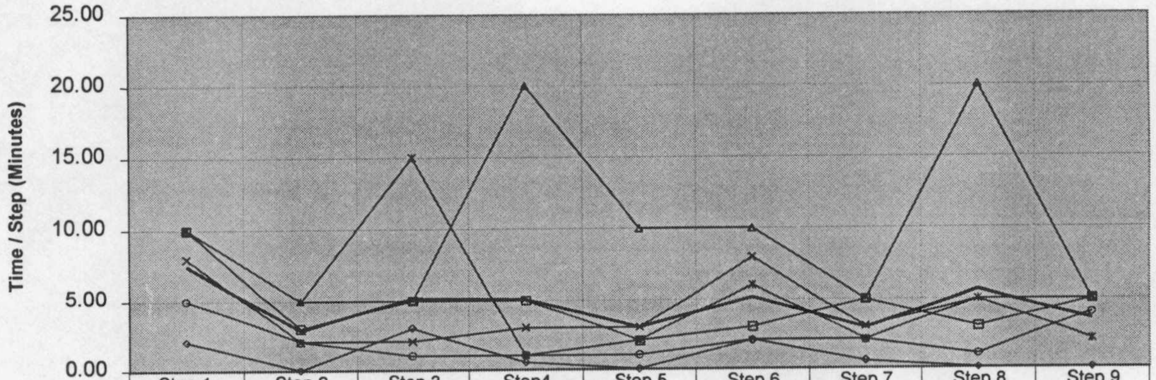
The estimates ignore the time it takes to select the device and assume that the user has enough knowledge about the system to know what to buy. For instance, the hard drive may be IDE, EIDE, SCSI, or USB and one or all may work Several of the people that responded also included a horror story related to their hardware experience

Table 6.1. Installing a second IDE hard 1



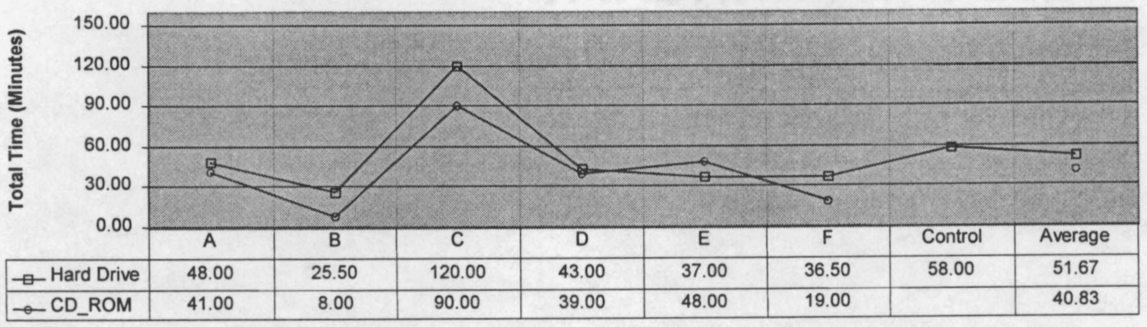
	Step 1	Step 2	Step 3	Step4	Step 5	Step 6	Step 7	Step 8	Step 9
□ A	10.00	5.00	10.00	5.00	2.00	3.00	5.00	3.00	5.00
◆ B	2.00	10.00	3.00	0.50	1.00	2.00	1.00	1.00	5.00
△ C	10.00	10.00	15.00	10.00	5.00	10.00	10.00	20.00	30.00
× D	8.00	10.00	5.00	1.00	2.00	8.00	2.00	2.00	5.00
✱ E	10.00	2.00	5.00	1.00	2.00	6.00	3.00	3.00	5.00
○ F	5.00	10.00	10.00	1.00	2.00	3.00	0.50	2.00	3.00
+ Measure	2.00	6.00	3.00	0.00	2.00	3.00	2.00	22.00	18.00
— Average	7.50	7.83	8.00	3.08	2.33	5.33	3.58	5.17	8.83

Table 6.2. Installing an internal CD-ROM 1



	Step 1	Step 2	Step 3	Step4	Step 5	Step 6	Step 7	Step 8	Step 9
□ A	10.00	3.00	5.00	5.00	2.00	3.00	5.00	3.00	5.00
◆ B	2.00	0.00	3.00	0.50	0.00	2.00	0.50	0.00	0.00
△ C	10.00	5.00	5.00	20.00	10.00	10.00	5.00	20.00	5.00
× D	8.00	2.00	2.00	3.00	3.00	8.00	3.00	5.00	5.00
✱ E	10.00	5.00	15.00	1.00	2.00	6.00	2.00	5.00	2.00
○ F	5.00	2.00	1.00	1.00	1.00	2.00	2.00	1.00	4.00
— Average	7.50	2.83	5.17	5.08	3.00	5.17	2.92	5.67	3.50

Table 6.3. Total installation times. 1



	A	B	C	D	E	F	Control	Average
Hard Drive	48.00	25.50	120.00	43.00	37.00	36.50	58.00	51.67
CD_ROM	41.00	8.00	90.00	39.00	48.00	19.00		40.83

1. One user accidentally unseated the cable that runs from the CD-ROM to the sound card while installing a hard drive. The plug was not totally disconnected but rather it was barely connected resulting in intermittent behavior. It took several hours of frustration to find the problem.
2. A second user removed all the case screws when trying to remove the case. Unfortunately, the screws that held the power supply, among other things, were t accessible from the outside and were removed. This resulted in much more work to re-assemble the system.
3. A third respondent unknowingly bent one of the pins on the hard drive as he was plugging the ribbon cable. When the system was turned on, it would not boot. After much frustration, the problem was finally discovered but not after reseating all the cards and connectors and repeated rebooting.
4. The time given for the validation installation does not include an addition three hours spent trying to get the system to boot into Windows 95. The system would boot into DOS and into what is called "Safe Mode" Windows which is where none of the special drivers are loaded, networking is not loaded, and only the basic monitor is available. The problem was caused by the fact that the system being updated was several years old and already had an existing master IDE drive. The new drive, EIDE, was placed on the same ribbon with the old drive (as one would expect) but they would not work together.

The above experiences help to emphasize the point that we do not want to open the computer case. The last one illustrates what can go wrong when a proprietary connection is updated without regard to existing systems. The installation of a keyboard and monitor are assumed trivial. Connection of a monitor does not include the installation of a video card. While this task is not necessarily difficult, the wide array of choices and possible conflicts with the motherboard make this something not typically attempted by a typical user.

The following steps are required to install any device in a system running JiniOS and could be expected to be performed by any skill level in under ten minutes.

1. Attach power cable and Ethernet cable.
2. Turn device on.

3 Wait for driver to broadcast to the Jini server and then be broadcast to the computer

4 Select the device

We can see that there is a considerable advantage of JiniOS over the current methods just from the installation standpoint While the time required might not be substantially less for JiniOS, there is a significant advantage when it comes to the required skill level and the risks associated with opening the computer Also since all devices are installed exactly the same way with exactly the same connectors, installation into JiniOS is much less intimidating Additionally, some devices such as hard drives may be shared among several computers without having to change anything in the operating system, the idea of upgrading a collection of computers is much more attractive

All communication to the devices in JiniOS is via a standard Ethernet connection Below are listed the connections used today and their respective throughputs ("What is USB?")

1	Parallel Port (printer)	150 Kbps (kilobits per second)
2	Serial Port (mouse)	920 Kbps
3	PS/2 Port (keyboard)	9 6 Kbps
4	EIDE (internal drive)	6 6 – 16 Mbps
5	USB Port (any)	1 5 – 12 Mbps (Megabits per second)
6	SCSI Port (drive)	20 Mbps
7	Ethernet (any)	10 Mbps, 100 Mbps

We can see from this that standard Ethernet, at 10 to 100 Mbps and possibly 1 gigabit per second (Gbps) in the near future, can easily accommodate these devices with little special consideration The only exception to this may in the area of video. The monitor driver for this project assumed a simple text-only monitor which could easily be handled here, however, when dealing with graphical monitors there is a much higher volume of data that must be transmitted For example, if we assume a standard monitor with a resolution of 1280 by 1024 pixels and 16 million colors, we would have to transmit 31,457,280 bits of information for every frame Assuming a refresh rate of thirty frames per second means having to transmit 943,718,400 bits every second

The X window system developed for the Unix platform allows applications to display windows connected to remote hosts It does this using a client/server model and transmitting highly compressed commands rather than screens for information A similar technique would have to be employed to effectively replace an internally connected video card with an external, remote one That is the commands now being sent to the card would have to be transmitted via Ethernet to an external monitor/video device If this were done then JiniOS would be able to handle graphical video monitors also

6.2 Conclusions

This research was started because the task of installing hardware into a computer was too difficult I have shown that there exists a much simpler technique using Jini, which will not only make the task of installing new devices almost trivial but also remove the dependence of a device on an operating system These methods could be employed to allow the same hard drive to be used by a personal computer, a Unix workstation, an Apple computer and a large mainframe computer In addition, all four could use it at the same time This simpler technique is much more fault-tolerant as well as much less restrictive Since drivers are distributed with the devices, new devices can enter the market much more quickly

Jini redefines the term device In the Jini specification, the items distributed by a Jini server are services These may be software services instead of hardware ones For example, a very fast computer can distribute a service in which it agrees to perform a series of calculations Or a database can advertise services which allow clients to access its data In addition the Discovery / Join protocol allows access to mobile networks For instance a laptop computer could connect to the network installed on an airplane and take advantage of the faster processor or printer or mail or even food services Since the requirements of Jini are minimal, 48K of Java binaries, there is a large market for retro-fitting existing devices The iPic is a complete single chip computer that runs a complete implementation of a TCP/IP stack and an HTTP web-server which can be coupled with a static RAM chip to deliver the necessary driver files ("Ipic – A Match Head Sized Web-Server") The iButton is an inexpensive 16 mm computer chip that can be configured to contain 64 Kbytes of RAM or a complete Java Virtual machine that is JavaCard™ compliant (iButton Overview) Cases that contain a small CPU, some RAM, and an Ethernet chip could easily be manufactured These could be then be attached to existing hardware and used in any Jini network Some consideration must be given as to how to update drivers that are already installed in hardware This could mean that a device might have a new driver loaded to it, which it stores in addition to the original In this way, a manufacturer could sell software updates or enhancements for devices it has already distributed

The quickly escalating public interest in "plug-and-play" devices ensures that some changes are imminent. Days after the release of Jini, Microsoft released Universal Plug and Play (www upnp org) which is their attempt to influence the device market ("Universal Plug and Play Background") UPNP is very similar to Jini in that it relies on a network and a universal server to distribute objects but is uses XML to transport them Unlike Jini UPNP does not standardize the language of the drivers and may result in having to distribute binaries based on operating system Similarly, The Salutation consortium (www salutation org) has released a specification for a service discovery and service management architecture similar to Jini but not based on Java It is independent of network transport, hardware platform, and operating system software and supports standard Internet and other message formats Unfortunately, the final solution will be all of the above, i.e., whichever method, or methods, are adopted, it will have to be able to accommodate the others There will never be a clear winner

Bibliography

Adjie-Winoto, William, Elliot Schwartz, Hari Balakishnan, and Jeremy Lilley, "The Design and Implementation of an Intentional Naming System", Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99), December 1999

Barham, P , M Hayter, D McAuley and I Pratt, "Devices on the Desk Area Network", IEEE Journal on Selected Areas in Communication, May 1995

"boot", http //www.whatis com/boot htm, March 2000

"Class ServiceID Man Page", Jini Technology version 1 0 API

Coffman, Edward, Jr and Peter Denning, Operating Systems Theory, Prentice-Hall, Inc, 1973

Dertouzos, Michael, "The Future of Computing", Scientific American, August 1999

"Discovery Devices and Services in Home Networking", IBM, Inc., http //www-3 ibm com/pvc/tech/networking html, March 2000

"DOS Batch Language A Personal View", http //www maem umr edu/~batch, June 1999

Edwards, W Keith, Core Jini, The Sun Microsystems Press, 1999

Flanagan, David, Java in a Nutshell, 2nd Edition, O'Reilly & Associates, 1997

Flanagan, David, Jim Farley, William Crawford, Kris Magnusson, Java Enterprise in a Nutshell, O'Reilly & Associates, 1999

Freeman, Eric, Susanne Hupfer, Ken Arnold, JavaSpaces Principles, Patterns, and Practice, Addison Wesley, 1999

Gosling, James, "A Brief History of the Green Project", http //java sun com/people/jag/green/index html, April 2000

Gronvall, Bjorn, Assar Westerland, and Stephen Pink, "The Design of a Multicast-based Distributed File System"" Operating Systems Review Conference, Winter 1998, pp 251-264

Harold, Elliotte, Java Network Programming, O'Reilly & Associates, 1997

Hayter, Mark and Derek McAuley, "The Desk Area Network", Operating Systems Review, Volume 25, Number 4, October 1991

Hunt, Craig, TCP/IP Network Administration, 2nd Edition, O'Reilly & Associates, 1998

"iButton Overview", http //www ibutton com/ibuttons/index html, October 1999

"Inferno", Lucent Technologies, http //inferno bell-labs com/inferno, April 2000

"Inferno la Commedia Interattiva", Lucent Technologies, http //inferno bell-labs com/inferno/infernosum html, April 2000

"IPic – A Match Head Sized Web-Server", http //www-ccs cs umass edu/~shri/iPic html, February 2000

"Jini™ Architecture Specification", Sun Microsystems, 1999

"Jini™ Device Architecture Specification", Sun Microsystems, 1999

"Jini™ Discovery and Join Specification", Sun Microsystems, 1999

"Jini™ Discovery Utilities Specification", Sun Microsystems, 1999

"Jini™ Distributed Leasing Specification", Sun Microsystems, 1999

"Jini™ Entry Specification", Sun Microsystems, 1999

"Jini™ Entry Utilities Specification", Sun Microsystems, 1999

"Jini™ Lookup Service Specification", Sun Microsystems, 1999

"Jini™ Technology Helper Utilities And Services Specification", Sun Microsystems, 1999.

"Jini Technology's Four-Year History",
http //www abcomp be/news/events/receptions/abny99/history html, February 2000

Kelly, Kevin and Spencer Reiss ,"One Huge Computer", Wired, v6 08, August 1998, pp 128-182

Knight, Richard, "Hard Disk Guide – Hard Disk Interfaces",
http //www makeitsimple com/articles/hdguide/hd_guide3 htm, March 2000

Lamb, Charles "JINI Net Intelligence a Your Command", Software Development, Volume 8, Number 2, February 2000

Montfort, Nick, "Breaking Protocol", Wired, v7 12, December 1999, pp 344-347

Mull, Allison and Tobin Maginnis, "Evolutionary Steps Toward a Distributed Operating Systems Theory and Implementation", Operating Systems Review, Volume 25, Number 4, October 1991

Newman, Terry, "Batch Guide", http //www nc5 infi.net/~wnewton/batch/batguide html, June 1999

Norman, Donald A., The Invisible Computer, The MIT Press, 1998

Pascoe, Robert, "Scheme 'discovers' networked services", EETimes com,
http //www eetimes com/story/OEG20000110S0027, April 2000

Pike, Rob, et al, "Plan 9 from Bell Labs", Bell Labs, http //plan9 bell-labs com/plan9/doc/9 html, March 2000

Pinto, Hugo José, "JINOS The Impromptu Operating System", CITAI Polytechnic Institute of Setúbal, Portugal, http //developer jini org/exchange/users/hugojpinto/JINOSPaper html, March 2000

Plank, James, Henri Casanova, Jack Dongarra, and Terry Moore, "Netsolve An Environment for Deploying Fault-tolerant Computing", FastAbstracts Session, FTCS-28 28th International Symposium on Fault-tolerant Computing, June 1998

Quinnell, Richard, "USB a neat package with a few loose ends",
http //www ednmag com/reg/1996/102496/df_01 htm, March 2000

Rathbone, Andy, “Upgrading & Fixing PCs for Dummies® 4th Edition”, IDG Books Worldwide, 1998

Rekesh, John, “UPnP, Jini and Salutation—A look at some popular coordination frameworks for future networked devices”, California Software Laboratories, http //www cswl com/whitepr/tech/upnp html, March 2000

Risley, David, “BIOS Guide”, http //www pcmech com/bios/index htm, March 2000

Ryan, Stein, “Synchronization in Portable Device Drivers”, Operating Systems Review, Volume 33, Number 1, January 1999

Schmidt, Brian, Monica Lam, and Duane Northcutt, “The Interactive Performance of SLIM a Stateless, Thin-client Architecture, Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP’99), December 1999

Sirer, Emin, Robert Grimm, Arthur Gregory, and Brian Bershad, “Design and Implementation of a Distributed Virtual Machine for Networked Computers”, Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP’99), December 1999

“Universal Plug and Play Background”, http //www upnp com/resources/UpnPbkgnd htm, March 2000

Vahdat, Amin, Thomas Anderson, Michael Dahlin, David Culler, Eshwar Belani,Paul Eastham, Chad Yoshikawa, “WebOS Operating System Services for Wide Area Applications”, The Seventh IEEE Symposium on High Performance Computing, July 1998

Van Meter, Rodney, “A Brief Survey of Current Works on Network Attached Peripherals”, Operating Systems Review, Volume 30, Number 1, January 1996

Waldo, Jim, Geoff Wyant, Ann Wollrath and Sam Kendall, “A note on Distributed Computing”, Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc , November 1994

Waldo, Jim, “The JINI Architecture for Network-Centric Computing”, Communications of the ACM, Volume 42, number 7, July 1999

Walker, Janice and Todd Taylor, The Columbia Guide to Online Style, Columbia University Press, 1998

“What is BIOS (Mini-FAQ)”, http //www sysopt com/biosdef.html, March 2000

“What is USB?”, Road Warrior Tips, Mobile and Wireless Guide, IGo Corporation, C 21 5

“What is USB and will it play a role in the 21st century?”,
http //www itweb co.za/sections/techforum/2000/0002090759 asp, March 2000

Appendices

Appendix A: RMI Sample

```
import java rmi.*;

public interface Hello extends Remote {
   public String sayHello() throws java.rmi.RemoteException;
}
```

Listing A 1 Interface

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject implements Hello {
   public HelloImpl() throws RemoteException {
      super();
   }
   public String sayHello() throws RemoteException {
      return("Hello World!");
   }
   public static void main(String[] args) {
      System.setSecurityManager(new RMISecurityManager());
      try {HelloImpl h=new HelloImpl();
           Naming.rebind("hello",h);
           System.out.println("Hello Server Ready.");
      } catch (RemoteException re) {
         System.out.println("RemoteException: " + re);
      } catch (MalformedURLException me) {
         System.out.println("MalformedURLException. " + me);
      }
   }
}
```

Listing A 2 Server.

```
import java rmi.*,

public class HelloClient {
   public static void main(String args[]) {
      System.setSecurityManager(new RMISecurityManager());

      try {Hello h = (Hello)Naming.lookup("hello");
           String message = h sayHello();
           System out.println("HelloClient: " + message),
      } catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

Listing A 3 Client

Appendix B Specification for Service ID

A universally unique identifier (UUID) for registered services A service ID is a 128-bit value Service IDs are intended to be generated only by lookup services, not by clients

The most significant long can be decomposed into the following unsigned fields

0xFFFFFFFF00000000	time_low
0x00000000FFFF0000	time_mid
0x000000000000F000	version
0x0000000000000FFF	time_hi

The least significant long can be decomposed into the following unsigned fields

0xC000000000000000	variant
0x3FFF000000000000	clock_seq
0x0000FFFFFFFFFFFF	node

The variant field must be 0x2 The version field must be either 0x1 or 0x4 If the version field is 0x4, then the most significant bit of the node field must be set to 1, and the remaining fields are set to values produced by a cryptographically strong pseudo-random number generator If the version field is 0x1, then the node field is set to an IEEE 802 address, the clock_seq field is set to a 14-bit random number, and the time_low, time_mid, and time_hi fields are set to the least, middle and most significant bits (respectively) of a 60-bit timestamp measured in 100-nanosecond units since midnight, October 15, 1582 UTC ("Class ServiceID Man Page")

Appendix C: JiniOS Server Sample - Keyboard

```
package keyboard;
import java.io.*;

public interface KeyboardServiceInterface {
   public void open()  throws IOException;
   public void close() throws IOException;
   public  int read()  throws IOException;
}
```

Listing C 1 Keyboard Interface

```
package keyboard;
import java.io.*;
import java.net.*;

public class KeyboardService implements Serializable,
keyboard.KeyboardServiceInterface{
   int port;
   String host;
   Socket socket;
   DataInputStream  get;
   DataOutputStream put;

   public KeyboardService() {}

   public KeyboardService(String host, int port) {
      this.host = host;
      this.port = port;
   }

   public void open() throws IOException {
      socket = new Socket(host,port);
      System.out.println("Connection established with "+socket);

      get = new DataInputStream (socket getInputStream ());
      put = new DataOutputStream(socket.getOutputStream());

      Lease lease = new Lease(5);
      lease.start();
   }

   public void close() throws IOException {
      put.write((byte)0);   put.flush(),
      socket.close(),
      socket = null;
   }

   public int read() throws IOException {
      return((int)get.readByte());
   }

   class Lease extends Thread {
      long lease=5000;
      public Lease(int lease) {this.lease = lease*1000;}
      public void run() {
         while (true) {
            try {Thread.sleep(lease);
                 if (socket == null) break;
                 put.write((byte)1);
                 put flush();}
            catch(Exception e) {e.printStackTrace();}
         }
      }
   }
}
```

Listing C 2 Keyboard Driver

```
package keyboard;

import java.io.*;

public class ServiceWrapper extends common.BasicService {
    static int port = 3493;

    public ServiceWrapper() throws IOException,
                                   ClassNotFoundException {
        super(port);
    }

    public Object getProxy(String host, int port) {
       return new KeyboardService(host,port);
    }

    public void run() {
        while (true) try {Thread.sleep(Long.MAX_VALUE);}
                     catch (InterruptedException ex) {}
    }

    protected void register() throws IOException {
        String product[] = {"Keyboard - Network accessable keyboard",
                            "OKBye, Inc.",
                            "OKBye, Inc.",
                            "v0.9b",
                            "132-Key Standard",
                            "000000000"};
        register(product);
    }

    public static void main(String[] args) {
        try {ServiceWrapper wrapper = new ServiceWrapper();
             new Thread(wrapper).start();}
        catch (Exception ex) {ex.printStackTrace();}
    }
}
```

Listing C.3. Keyboard Service

```
package common,

import java.io.*;
import java.net.*;

import net.jini.core.lookup.ServiceID;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.discovery.LookupLocator;
import net.jini.core entry.Entry;

import net.jini.lookup.entry.*;
import com.sun.jini.lookup.JoinManager;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lookup entry.BasicServiceType;

import java.rmi.RMISecurityManager;


public abstract class BasicService implements Runnable {
    protected JoinManager join = null;
    protected String serFile   = "serialization.storage";
    protected Object proxy     = null;
    protected String host      = "localhost";

    static class PersistentData implements Serializable {
        ServiceID serviceID;
        Entry[] attrs;
        String[] groups;
        LookupLocator[] locators;

        public PersistentData() {}
    }

    class IDListener implements ServiceIDListener {
        public void serviceIDNotify(ServiceID serviceID) {
            System.out.println("Got service ID " + serviceID);
            PersistentData state = new PersistentData();
            state.serviceID = serviceID,
            state.attrs     = join.getAttributes();
            state.groups    = join.getGroups();
            state.locators  = join getLocators();

            try {writeState(state);}
            catch (IOException ex) {
                System.err println("Couldn't write to file: " +
                                   ex.getMessage());
                ex.printStackTrace();
                join.terminate();
                System.exit(1);
            }
        }
    }
```

Listing C 4 Service Boilerplate

```
public abstract Object getProxy(String host, int port);

public BasicService(int port) throws IOException,
                                     ClassNotFoundException {
    if (System.getSecurityManager() == null)
       System.setSecurityManager(new RMISecurityManager());

    try {host = InetAddress.getLocalHost().getHostName();}
    catch (UnknownHostException e) {e.printStackTrace();}

    proxy = getProxy(host,port);

    File file = new File(serFile);
    if (file.exists()) reregister();
    else               register();
}

public void run() {
    while (true) try {Thread.sleep(Long.MAX_VALUE);}
                 catch (InterruptedException ex) {}
}

protected void register() throws IOException {
    String product[] = {"PRODUCT",
                        "MANUFACTURER",
                        "VENDOR",
                        "VERSION",
                        "MODEL",
                        "SERIALNUMBER"};
    register(product);
}

protected void register(String[] info) throws IOException {
    if (join != null)
       throw new IllegalStateException("Wrapper already started.");

    System.out.println("Starting...");

    Entry[] attributes = new Entry[]{new ServiceInfo(info[0],
                                                     info[1],
                                                     info[2],
                                                     info[3],
                                                     info[4],
                                                     info[5]),
                              new BasicServiceType("Service")};

    join = new JoinManager(proxy,            // Object to register
                           attributes,       // Entry attributes
                           new IDListener(), // Service ID listener
                           null);            // Lease manager
}
```

Listing C 4 Service Boilerplate (continued)

```
    protected void reregister() throws IOException,
                                       ClassNotFoundException {
        if (join != null)
           throw new IllegalStateException("Wrapper already started.");

        PersistentData state = readState();

        System.out.println("Restarting: old id is "+state.serviceID);
        join = new JoinManager(state.serviceID, // Service ID
                               proxy,           // Object to register
                               state.attrs,     // Entry attributes
                               state.groups,    // Groups to join
                               state.locators,  // Lookup locators
                               null);           // Lease manager
    }

    protected void writeState(PersistentData state) throws IOException{
        ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream(serFile));
        out.writeObject(state);
        out.flush();
        out.close();
    }

    protected PersistentData readState() throws IOException,
                                             ClassNotFoundException{
        ObjectInputStream in = new ObjectInputStream(
                                  new FileInputStream(serFile));
        PersistentData state = (PersistentData)in.readObject();
        in.close();
        return state;
    }
}
```

Listing C 4 Service Boilerplate (continued)

```
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <stdlib h>
#include <string.h>

#include <process.h>
#include <winsock.h>

#include "thread.h"

static unsigned short port=3493;

int lease=5;
time_t theTime=0;
unsigned long thread[]={0,0,0};

void ByteSwapAddr(unsigned int *word) {
   *word =  (*word&0xFF000000)>>24 |
            (*word&0x00FF0000)>> 8 |
            (*word&0x0000FF00)<< 8 |
            (*word&0x000000FF)<<24;
}

float ByteSwapFlt(float word)
   {ByteSwapAddr((unsigned int*)&word);   return(word);}

int ByteSwapInt(int word)
   {ByteSwapAddr((unsigned int*)&word);   return(word);}

unsigned int ByteSwapUns(unsigned int word)
   {ByteSwapAddr((unsigned int*)&word),   return(word);}

void LeaseThread(void *arg)
{
   SOCKET *client=(SOCKET*)arg;

   theTime = time(0) + lease;
   while (1) {sleep(lease);
              if ((theTime+lease-1) <= time(0)) break;}
   printf("Client has died -- connection terminated.\n");
   thread[0] = 0;
}

void SignalThread(void *arg)
{
   char stream[8]={0};
   SOCKET *client=(SOCKET*)arg;

   while (1) if (recv(*client,stream,1,0) > 0) {
                  if (*stream == 0) break;
                  else theTime = time(0) + lease;
             }
```

Listing C 5 Keyboard Driver

```
   printf("Connection terminated by client request.\n");
   thread[1] = 0;
}

void ClientThread(void *arg)
{
   int count;
   SOCKET *client=(SOCKET*)arg;
   for (count=1; 1; count++) {int key=_getch();
                              send(*client,(char*)(&key),1,0);}
}

int main(int argc, char** argv)
{
   int status;
   WSADATA WSAData;
   if ((status=WSAStartup(MAKEWORD(1,1),&WSAData)) == 0) {
      char hostname[32];
      SOCKET server;
      SOCKADDR_IN serverAddr;

      printf("Description:%s   Status:%s\n",
         WSAData.szDescription,
         WSAData.szSystemStatus);

      gethostname(hostname,sizeof(hostname));
      printf("Host: %s\n",hostname);

      serverAddr.sin_family      = AF_INET;
      serverAddr.sin_port        = htons(port);
      serverAddr.sin_addr.s_addr = INADDR_ANY;

      if ((server=socket(AF_INET,SOCK_STREAM,0)) == INVALID_SOCKET)
         printf("Socket() failed.\n");
      else if ((bind(server, (struct sockaddr *)&serverAddr,
                  sizeof(serverAddr))) == SOCKET_ERROR)
         perror("bind");
      else if (listen(server,10) == -1) perror("listen");
      else while(1) {
         SOCKET client;
         SOCKADDR_IN clientAddr,
         int size=sizeof(clientAddr);
         printf("...waiting on a client connection...\n");
         if ((client=accept(server,
                           (struct sockaddr *)&clientAddr,
                           &size)) == -1)
            perror("accept");
         else {printf("...got a client connection -- press key\n");
               thread[0] = _beginthread(LeaseThread,
                                        (unsigned)0,
                                        (void*)(&client));
               thread[1] = _beginthread(SignalThread,
```

Listing C 5 Keyboard Driver (continued)

```
                                        (unsigned)0,
                                        (voıd*)(&clıent));
                thread[2] = _beginthread(ClientThread,
                                        (unsıgned)0,
                                        (voıd*)(&client));

                WaıtForMultipleObjects(3, (HANDLE*)thread,
                                      FALSE, INFINITE);
                if (thread[0]) TerminateThread((HANDLE)thread[0],0);
                ıf (thread[1]) TerminateThread((HANDLE)thread[1],0);
                ıf (thread[2]) TerminateThread((HANDLE)thread[2],0);
                closesocket(client);}
        }
    }
    else prıntf("WSA Error %d\n",status),

    prıntf("Finı\n"),
    fgetc(stdin);
    return(0);
}
```

Lıstıng C 5 Keyboard Drıver (contınued)

Appendix D: JiniOS Client Sample - Keyboard

```
package keyboard;
import java.io.IOException,

public class KeyboardClient extends common.BasicClient {

   public KeyboardClient() throws IOException {
      super(KeyboardServiceInterface.class);
   }

   public KeyboardServiceInterface Keyboard() {
      return((KeyboardServiceInterface)getClient(0));
   }

   public KeyboardServiceInterface Keyboard(int index) {
      return((KeyboardServiceInterface)getClient(index));
   }

   public static void main(String args[]) {
      try {
         KeyboardClient client = new KeyboardClient();

         client.waitForService();
         System.out println("Available      "+client.available());
         System.out.println("Name          :"+client.name());
         System.out.println("Model         :"+client.model());
         System.out.println("Version       :"+client.version());
         System.out.println("Vendor        ."+client.vendor());
         System.out.println("Manufacturer  :"+client.manufacturer());
         System.out.println("Serial Number :"+client.serialNumber());

         KeyboardServiceInterface keyboard = client.Keyboard();
         keyboard.open();

         try {
            while (true) {int key = keyboard.read();
                          System.out.print(key==13 ? '\n' : (char)key);
                          if (key == 'q') break;}
         } catch (IOException e) {e.printStackTrace();}

         keyboard.close();
      } catch (IOException ex) {System.out.println("Couldn't create " +
                                     client: " + ex.getMessage());}
   }
}
```

Listing D 1 Keyboard Client

```
package common;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;

import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;

import net.jini.core.entry.*;
import net.jini.core.event.*;
import net.jini.core.lookup.*;

import net.jini.lookup.entry.*;
import com.sun.jini.lease.LeaseRenewalManager;
import com.sun.jini.lookup.entry.BasicServiceType;

import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class BasicClient {
   protected Vector service = new Vector();
   protected ServiceTemplate template;
   protected LookupDiscovery disco;
   protected LeaseRenewalManager leaseMgr = new LeaseRenewalManager();
   protected Listener eventListener;

   class Discoverer implements DiscoveryListener {
      public void discovered(DiscoveryEvent ev) {
         ServiceRegistrar regs[] = ev.getRegistrars();
         for (int i=0; i<regs.length; i++) addService(regs[i]);
      }

      public void discarded(DiscoveryEvent ev) {
         ServiceRegistrar regs[] = ev.getRegistrars();
         for (int i=0; i<regs.length; i++) removeService(regs[i]);
      }
   }

   class Listener extends UnicastRemoteObject
                  implements RemoteEventListener{
       Listener() throws RemoteException {super();}
       public void notify(RemoteEvent ev) {
          if (ev instanceof ServiceEvent)
             addService((ServiceRegistrar)ev.getSource());
       }
   }
```

Listing D 2 Client boilerplate

```
public BasicClient(Class type) throws IOException {
   if (System.getSecurityManager() == null)
       System.setSecurityManager(new RMISecurityManager());

    Class serviceTypes[] = {type};
    template = new ServiceTemplate(null, serviceTypes, null);

    disco = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    disco.addDiscoveryListener(new Discoverer());
    eventListener = new Listener();
}

protected void addService(ServiceRegistrar serviceRegistrar) {
   EventRegistration eventRegistration,
   try {eventRegistration =
           serviceRegistrar.notify(template,
              ServiceRegistrar.TRANSITION_MATCH_NOMATCH |
              ServiceRegistrar.TRANSITION_NOMATCH_MATCH |
              ServiceRegistrar.TRANSITION_MATCH_MATCH,
              eventListener, null, 10 * 60 * 1000);
        leaseMgr.renewUntil(eventRegistration.getLease(),
                            Long.MAX_VALUE, null);
   } catch (RemoteException e) {disco.discard(serviceRegistrar);
                                 e.printStackTrace();
                                 return;}

   ServiceMatches matches = null;
   try {if ((matches = serviceRegistrar.lookup(template,
                             Integer.MAX_VALUE))== null)
            System.err.println("No matching service ");
        else {System.out println("Got a matching service.");
              for (int i=0; i<matches.totalMatches; i++) {
              if (service.contains(matches.items[i]))
                 service.remove(matches.items[i]);
                 service.add(matches.items[i]);
              }}
   } catch (Exception e) {e.printStackTrace();}
}

protected void removeService(ServiceRegistrar serviceRegistrar) {
   ServiceMatches matches;
   try {if ((matches = serviceRegistrar.lookup(template,
                             Integer.MAX_VALUE)) != null)
            for (int i=0; i<matches.totalMatches; i++)
               if (service.contains(matches.items[i]))
                  service.remove(matches.items[i]);
   } catch (Exception e) {e.printStackTrace();}
}
```

Listing D 2 Client boilerplate (continued)

```
  public void waitForService() {
     while (available() == 0) try {Thread sleep(1000);}
                              catch (InterruptedException ex) {}
  }

  private String attribute(int index, String attributeName) {
     if (available() <= index) return(null);
     ServiceItem serviceItem = (ServiceItem)service.get(index);
     Entry[] attributes      = serviceItem.attributeSets;
     for (int i=0; i<attributes.length; i++)
        if (attributes[i] instanceof ServiceInfo)
           if      (attributeName.equalsIgnoreCase("Name"        ))
              return(((ServiceInfo)attributes[i]).name);
           else if (attributeName.equalsIgnoreCase("Model"       ))
              return(((ServiceInfo)attributes[i]).model);
           else if (attributeName.equalsIgnoreCase("Version"     ))
              return(((ServiceInfo)attributes[i]).version);
           else if (attributeName.equalsIgnoreCase("Vendor"      ))
              return(((ServiceInfo)attributes[i]).vendor);
           else if (attributeName.equalsIgnoreCase("Manufacturer"))
              return(((ServiceInfo)attributes[i]).manufacturer);
           else if (attributeName.equalsIgnoreCase("SerialNumber"))
              return(((ServiceInfo)attributes[i]).serialNumber);
     return(null);
  }

  public int available () {
     return(service.size());
  }
  public String name    (int index){return(attribute(index,"Name"));}
  public String model   (int index){return(attribute(index,"Model"));}
  public String version (int index)
     {return(attribute(index,"Version"));}
  public String vendor (int index)
     {return(attribute(index,"Vendor"));}
  public String manufacturer (int index)
     {return(attribute(index,"Manufacturer"));}
  public String serialNumber (int index)
     {return(attribute(index,"SerialNumber"));}
  public String name         ()           {return(name        (0));}
  public String model        ()           {return(model       (0)),}
  public String version      ()           {return(version     (0));}
  public String vendor       ()           {return(vendor      (0));}
  public String manufacturer ()           {return(manufacturer(0));}
  public String serialNumber ()           {return(serialNumber(0));}

  public Object getClient(int index) {
     waitForService();
     if (available() <= index) return(null);
     ServiceItem serviceItem = (ServiceItem)service.get(index);
     return(serviceItem.service);
  }
}
```

Listing D 2 Client boilerplate (continued)

Appendix E: JiniOS Boot Batch File

```
@echo off

REM Set hostname
call \java\JiniProjects\scripts\hostname
echo Host: %HOSTNAME%

REM Set base variables
set JINI=\Java\Jini\files\jini1_0

REM Set the rest
set LOOKUP_POLICY=%JINI%\example\lookup\policy
set BROWSER_POLICY=%JINI%\example\browser\policy
set JARFILE=%JINI%\lib\reggie.jar
set CODEBASE=http://%HOSTNAME%:8080/reggie-dl.jar
set LOGDIR=\Java\JiniArea\reggie_log
set GROUP=public
set BROWSER=com.sun.jini.example.browser.Browser

@echo on
REM Start HTTP Server
start \java\jdk1.2.1\bin\java
      -jar %JINI%\lib\tools.jar
      -port 8080
      -dir %JINI%\lib
      -verbose

REM Start RMI Activation Daemon
deltree /y log
start \java\jdk1.2.1\bin\rmid

REM Wait for RMI Activation Daemon to finish loading
\java\jdk1.2.1\bin\java
      -cp \Java\JiniTutorial\CoreJini\examples-util.jar
      com.sun.jini.example.books.RMIDWait

REM Start Lookup Service
deltree /y %LOGDIR%
start \java\jdk1.2.1\bin\java
      -Djava.security.policy=%LOOKUP_POLICY%
      -jar %JINI%\lib\reggie.jar
      %CODEBASE% %LOOKUP_POLICY% %LOGDIR% %GROUP%

echo off

:fini
```

Listing E 1 Start Jini System

```
@echo off


REM Set hostname
call \java\JiniProjects\scripts\hostname
echo Host: %HOSTNAME%

start \java\jdk1.2.1\bin\java
      -jar \Java\Jini\files\jini1_0\lib\tools.jar
      -port 8085
      -dir \Java\JiniArea\service-dl
      -verbose

set STDCP=\Java\Jini\files\jini1_0\lib\jini-core.jar;
      \Java\Jini\files\jini1_0\lib\jini-ext.jar;
      \Java\Jini\files\jini1_0\lib\sun-util.jar

@echo on
start ..\Native\keyboard
\java\jdk1.2.1\bin\java
      -cp %STDCP%;\Java\JiniArea\service
      -Djava.rmi.server.codebase=http://%HOSTNAME%:8085/
      -Djava.security.policy=\Java\JiniArea\policy
      keyboard.ServiceWrapper
```

Listing E 2 Start Keyboard Service

```
@echo off


set STDCP=\Java\Jini\files\jini1_0\lib\jini-core.jar;
      \Java\Jini\files\jini1_0\lib\jini-ext.jar;
      \Java\Jini\files\jini1_0\lib\sun-util.jar

@echo on
\java\jdk1.2.1\bin\java
      -cp %STDCP%;\Java\JiniArea\client
      -Djava.security.policy=\Java\JiniArea\policy
      keyboard.KeyboardClient
```

Listing E 3 Start Keyboard Client

Appendix F: Installation Survey

Adapted from Upgrading & Fixing PCs for Dummies, 4th Edition by Andy Rathbone (Rathbone)

Adding a second IDE/EIDE hard drive to a PC running Windows 95

_____1 Turn off the computer, unplug it, and remove the case

a) Turn off the computer, monitor, and peripherals (modem, CD-ROM, and so on) Make sure that everything attached to your computer is turned off and unplugged
b) Unplug your computer Unplug your computer's power cord from the wall
c) Remove the screws from the PC's back or outside edges The older PC, the more screws you'll find.
d) Slide off the cover On some computers, the cover slides toward the front while on other computers, the cover lifts up and off

_____2 Since you are adding a second drive, you might need to move a jumper to change it to a slave drive (as opposed to a master) You have to check the drive's manual about how to do this Also be careful to touch the case to discharge any electrostatic charge before handling the drive

_____3 Slide the new drive into a vacant bay If the new drive is smaller than the bay you need to add rails or mounting brackets to make the hard drive fit the available space For example if you are mounting a 3 5-inch drive in a 5 25-inch bay, you will need to attach mounting brackets to each side of the drive so that it can be mounted in the bay The brackets and additional screws will sometimes be supplied with the drive, otherwise, you will have to buy them

_____4 Attach two cables to the hard drive (look at the cables attached to the main drive for help) Make sure that no cables have come loose either from the motherboard or other devices

a) Ribbon cable (wide cable) Since this is a second drive, the ribbon cable running to the main drive should have another connector on it The connector is keyed (has a notch or ridge that must match the ridge or notch on the drive) and must be attached correctly Typically one side of the cable will be marked (striped or colored) This it the side that should line up with the pin marked number 1 on the drive
b) Power cable (smaller cable) The computer should have spares (unattached) of these that can be used These are also keyed and will only fit one way

_____5 Attach the drive to the bay with all four screws

_____6 Replace the cover, attach all the peripherals, plug everything in, and turn on the computer

_____7 Configure the CMOS for the new drive This is where you tell the computer the details about the new drive This is accomplished in one of three ways

a) Some drives check the CMOS to see what hard drive your computer expects to find and then automatically mimic that drive (Plug-and-Play) If the computer is fairly recent and the drive is relatively new, this it the most common avenue
b) Others let you pick any hard drive that is listed in the computer's CMOS table
c) Still others require you to specify the drive's recommended cylinders, heads, and sectors These specifications are usually buried in the drive manuals

_____8 Partition the drive Open a DOS window and type **FDISK** Sometimes Windows will hang and you will have to reboot into DOS by pressing the F8 key while it is booting Select the option to change the current fixed drive to the new drive Create a single logical DOS partition, save, and exit Windows 95 will not allow the use of a drive larger than 2 Gbytes, so the drive must be partitioned into pieces smaller than 2 Gbytes before it can be formatted Windows 98 and Windows NT do not have this restriction. When you exit from **FDISK**, the computer must be rebooted

_____9 Format the drive After the computer completely reboots (from Step 8), the drive has to be formatted before it can be used Each partition created in Step 8 has to be formatted separately. If this is the second drive, its drive letter will most probably be D (and then E, F, G, and so on) This may cause problems with other disk devices (CD-ROM, Zip, etc), because their drive letters have been increased to accommodate the new drive The drive can be formatted in several ways

a Right-click the new drive in Windows Explorer and select format from the menu
b Right-click the drive in the "My Computer" window and select format
c Open a DOS window and type **FORMAT x:**, where x is the drive letter of the new drive

Installing an internal CD-ROM in a PC running Windows 95

_____1 Turn off the computer, unplug ıt, and remove ıts case· See Step 1 ın the prevıous procedure for ınstallıng a second hard drıve for complete steps

_____2 Plug the CD-ROM drıve's card ınto one of the avaılable slots and screw ıt down

_____3 Fınd an avaılable bay Remove the plate that covers openıng ın the case These are usually held ın place by plastıc tabs at eıther sıde of the plate Slıde the CD-ROM drıve ınto the bay from the front of the computer

_____4 Connect the cables Fırst connect the cable between the CD-ROM drıve and the card ınstalled ın Step 2 Consult the ınstallatıon manual supplıed wıth the drıve about where the cable attaches to the card and the drıve Next connect the power cable As wıth the hard drıve, there should be a free (unconnected) cable avaılable

_____5 Screw the drıve in place Some drıves attach wıth screws ın the sıdes or two screws ın the front

_____6 Replace the cover, attach all the peripherals, plug everythıng ın, and turn on the computer

_____7 Waıt for the computer to completely boot

_____8 Run the CD-ROM drıve's software If the drıve ıs Plug-and-Play, Wındows may eıther ınstall the new devıce automatıcally or ıt may prompt the user for a devıce drıver dısk or ıt may do nothıng In the last case, run the ınstall dısk provıded by the CD-ROM manufacturer to load the drıvers manually

_____9 The computer may have to be rebooted before the drıve can be used

Vita

Richard Clippard was born in Ft McClellan, Alabama on April 21, 1959 His father was serving in the Army most of Richard's early life so Richard attended schools throughout the southern United States and Germany He graduated from Franklin County High School in Winchester Tennessee in 1977 He received an Associate of Arts in Mathematics from Motlow State Community College, Tullahoma, Tennessee in 1979 He earned a Bachelor of Arts degree in Mathematics and a Bachelor of Arts degree in Computer Science from The University of Tennessee, Knoxville Tennessee in 1982 Richard entered The Master's program in Computer Science at The University of Tennessee Space Institute in 1996, officially receiving the Master's degree in August, 2000

He is presently employed as a senior programmer analyst at the Arnold Engineering Development Center in Arnold Air Force Base, Tennessee