MICROJINI: A SERVICE DISCOVERY AND DELIVERY INFRASTRUCTURE FOR PERVASIVE COMPUTING

By

DAVID ROGER NORDSTEDT

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING

UNIVERSITY OF FLORIDA

2001
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Abdelsalam (Sumi) Helal, for all his support and for giving me the opportunity to be a part of this exciting project. I would like to thank Dr. Doug Dankel and Dr. Beverly Sanders for sitting on my graduate committee. I feel very fortunate to have these members as my committee—they have taught my favorite and most beneficial classes at the University of Florida. All of them show a sincere concern for the welfare and education of their students.

I would also like to thank my fellow students Choonhwa Lee, Sam Coons, and Jinsuo Zhang for their work on this project. Choonhwa and Sam were part of the original project team and we all worked on the architecture together over a period of many months. Jinsuo took the time to explore the inner workings of VNC as well as the PNG libraries and added new functionality to the VNC server.

I would like to thank the Motorola Corporation for sponsoring our work, dubbed the JiniDEN project. They have provided the research team with funding as well as provided the i85s phones as physical network devices. They also provided valuable inside technical help for understanding the phone architecture, development platform, and limitations. Nextel Corporation has supplied us with network connectivity through their iDEN network for the duration of our project.

I would especially like to thank my wife, Mary, for all her encouragement, support, and patience during my last several years of continuing education.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................................................................................ iii

LIST OF TABLES ............................................................................................................. vii

LIST OF FIGURES ......................................................................................................... ix

ABSTRACT ....................................................................................................................... x

1 INTRODUCTION .......................................................................................................... 1
   Service Discovery and Delivery .................................................................................. 1
   µJini Motivation ......................................................................................................... 2
   Implementation .......................................................................................................... 3
   Thesis Organization .................................................................................................... 3

2 SERVICE DISCOVERY TECHNOLOGIES ................................................................. 5
   Jini .............................................................................................................................. 6
      Discovery ................................................................................................................ 7
      Service Lookup ..................................................................................................... 7
      Service Use ........................................................................................................... 8
      Error Handling .................................................................................................... 8
   UPnP .......................................................................................................................... 8
      Discovery ................................................................................................................ 9
      Service Lookup ..................................................................................................... 9
      Service Use .......................................................................................................... 10
      Error Handling ................................................................................................... 10
   Salutation ................................................................................................................... 10
      Discovery .............................................................................................................. 11
      Service Lookup and Use .................................................................................. 11
      Error Handling ................................................................................................... 11
   Jxta ........................................................................................................................... 11
   Compact .NET .......................................................................................................... 13
      Overview ............................................................................................................. 13
      Service Registration, Lookup, and Use ............................................................. 14
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Jxta Protocols ..................................................................</td>
<td>12</td>
</tr>
<tr>
<td>2. µJini Protocol Classes .........................................................</td>
<td>25</td>
</tr>
<tr>
<td>3. RFB Update Protocols ................................................................</td>
<td>32</td>
</tr>
<tr>
<td>4. RFB Protocol Client Messages ................................................</td>
<td>34</td>
</tr>
<tr>
<td>5. RFB Protocol Server Messages ................................................</td>
<td>35</td>
</tr>
<tr>
<td>6. Performance Data for iCoffee MIDlet .......................................</td>
<td>40</td>
</tr>
<tr>
<td>7. Performance Data for Shopping MIDlet .......................................</td>
<td>40</td>
</tr>
<tr>
<td>8. Performance Data for PhoneClient MIDlet ..................................</td>
<td>40</td>
</tr>
<tr>
<td>9. Performance Data for MapIt MIDlet ..........................................</td>
<td>41</td>
</tr>
<tr>
<td>10. Average Network Round-Trip Times ........................................</td>
<td>47</td>
</tr>
<tr>
<td>11. Test Case 1 Performance Data ................................................</td>
<td>48</td>
</tr>
<tr>
<td>12. Test Case 2 Performance Data ................................................</td>
<td>49</td>
</tr>
<tr>
<td>13. Test Case 3 Performance Data ................................................</td>
<td>50</td>
</tr>
<tr>
<td>14. Test Case 4 Performance Data ................................................</td>
<td>50</td>
</tr>
<tr>
<td>15. Test Case 5 Performance Data ................................................</td>
<td>51</td>
</tr>
<tr>
<td>16. Beginning of FramebufferUpdate Message ..................................</td>
<td>55</td>
</tr>
<tr>
<td>17. Second Stage of FramebufferUpdate Message ..............................</td>
<td>55</td>
</tr>
<tr>
<td>18. Appliance Object Description Value .......................................</td>
<td>56</td>
</tr>
<tr>
<td>19. Appliance Line Object ..........................................................</td>
<td>56</td>
</tr>
<tr>
<td>20. Appliance Rectangle Object ..................................................</td>
<td>56</td>
</tr>
</tbody>
</table>
21. Appliance Oval Object.................................................................................................... 56
22. Appliance Text Object .................................................................................................... 57
23. Appliance Button Object................................................................................................. 57
24. Appliance List Object...................................................................................................... 57
25. Text String Object........................................................................................................... 57
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Probable .NET Service Discovery Scenario</td>
<td>14</td>
</tr>
<tr>
<td>2. Motorola i85s Phone</td>
<td>16</td>
</tr>
<tr>
<td>3. Palm m505 Handheld Device</td>
<td>18</td>
</tr>
<tr>
<td>4. Compaq iPAQ 3650 PocketPC</td>
<td>19</td>
</tr>
<tr>
<td>5. p503is iMode Capable Phone</td>
<td>20</td>
</tr>
<tr>
<td>6. Overall µJini Architecture</td>
<td>22</td>
</tr>
<tr>
<td>7. µJini Communication Example</td>
<td>29</td>
</tr>
<tr>
<td>8. i85s Keypad Layout</td>
<td>37</td>
</tr>
<tr>
<td>9. Service Startup Time Comparison</td>
<td>41</td>
</tr>
<tr>
<td>10. Phone Screens Used in Performance Tests</td>
<td>46</td>
</tr>
<tr>
<td>11. Original Test Images as Seen on Server</td>
<td>46</td>
</tr>
<tr>
<td>12. Test Data by Update Data Size</td>
<td>52</td>
</tr>
<tr>
<td>13. Test Data by Round-Trip Time</td>
<td>52</td>
</tr>
<tr>
<td>14. Possible Appliance API Server Circuit Components</td>
<td>59</td>
</tr>
<tr>
<td>15. Possible Home Network with Appliance Server</td>
<td>60</td>
</tr>
<tr>
<td>16. Thermostat Display</td>
<td>61</td>
</tr>
</tbody>
</table>
As electronic circuits become smaller and more powerful in a parallel convergence, mobile computing devices are becoming more pervasive. This occurrence is creating a new business model within the computing industry. The model is one where users will pay for new and different services on their small computing device as they need them instead of purchasing a software package and installing it into the persistent storage area of the device. Users will also want mobile services that give them the same power of network interaction as their desktop computer. These services may include email, web browsing, map services, restaurant menus, entertainment news, videophone calls, and spreadsheet or word processing applications. With mobile networks there is another dimension called location-aware computing—service discovery should happen with respect to the area surrounding the user. Users will often be interested in services available within the geographical, spatial, or temporal dimension that is surrounding
them. When leaving a dimension, access to services in that particular dimension should disappear from the user’s point of view.

A particular problem arises because small devices do not typically have the resources that are available on average desktop computers and this restricts them from using current techniques such as Sun Microsystem's Jini architecture. Another problem area is that services may be written for many different platforms. Small devices do not typically have the ability to run programs designed for different platforms such as Solaris, Linux, Windows, or Java 2 applications.

The µJini idea promises to allow users of a small device such as a phone to discover remote or local services on the network and to automatically use these services. This thesis presents an architecture and a set of protocols for small devices to implement service discovery similar to the Jini specifications. We also design techniques that can be used for the service delivery after initial discovery. Any type of service can be found regardless of the service discovery protocol a service is using. In addition, the service can be returned to the client for use in a matter of seconds. A significant development of the research is the allowance for plug-in support and scalability for newer, more powerful devices while still catering to the smaller, resource-constrained devices. Smaller devices will always exist and perhaps be more prevalent than their larger computing counterparts in the near future. Another great value to service providers lies within the method in which a service is delivered. With a method shown in this thesis, the service provider can retain the binary code and thus prevent reverse-engineering tactics. In addition, because the client does not have a copy of the code, the service cannot be copied to another device for use by someone who has not purchased the service.
CHAPTER 1
INTRODUCTION

As electronic circuits become smaller and more powerful in a parallel convergence, mobile computing devices are becoming more pervasive. The business market for computing is moving into a world where users want services instead of applications on their computing device. With the proliferation of mobile devices such as cellular phones, applications once found only on desktop computers will be moved to smaller devices such as phones and Personal Digital Assistants (PDAs) for the convenience of the modern mobile computing user. With mobile devices, users will want to find new applications that are available to them as they travel from place to place. A user should not care whether the service is a physical program on the device or being executed on a server machine with only screen updates being displayed on the mobile device. This introductory chapter explains the main objective of this thesis, the main ingredients of the research, and a layout map for the rest of the thesis.

Service Discovery and Delivery

Service discovery has been implemented in many systems such as Sun’s Jini [1], the Microsoft-endorsed UPnP [2, 3], and Salutation [4, 5]. Each is implemented differently and with separate pros and cons [6, 7]. The similarity between these systems is to discover a service dynamically without knowing how the service is implemented. For example, if a user desires a map of the current location, a query should be made for a map service. A service discovery system would accept the request and return all
discovered implementations of map services to the user. The service would be responsible for providing the method call implementations and return values to interact with the service. Within mobile networks there is an additional dimension of location – service discovery should happen with respect to the local area of the user. When in a certain area, the user may only want services available within that localized area. When leaving an area, access to services in that location should disappear from the user’s view.

µJini Motivation

This µJini (pronounced micro-genie) idea promises to allow users of a small device such as a phone to discover local services on the network and to automatically use these services.

A Jini system already provides for the dynamic service discovery process desired, but requires serialization¹ and RMI² to be present on the client device. Serialization and RMI require more resources that are normally found on a small device. This is because additional resources are too costly for device manufacturers and consumers at the present. µJini is the name given to the architecture that has been designed to avoid the need for serialization and RMI. This project implements a simple messaging application programming interface (API) so that a small client and proxy server can communicate and interact in service discovery and delivery actions.

There has been work done by others in regard to connecting non-java devices to a Jini network, but this has been with respect to the service-side device and not the client-

¹ Serialization is a term used to reflect breaking an object into a byte stream, sending it over the network, and reconstituting it on the other side.

² RMI is Remote Method Invocation and is included with Java 2. This allows the capability of making remote method calls over a network; analogous to RPC in unix systems.
side device [8, 9]. The µJini project focuses on the client side of a Jini system and provides for a small device to use any Jini service available on a network.

Implementation

For the client platform, Motorola’s new i85s model phone has been chosen. This phone operates using a proprietary native operating system and is J2ME compliant.³ This project targets J2ME as the programming platform and therefore native phone OS details are not of importance. Java 2 Standard Edition (J2SE) [10] on a Solaris operating system (OS) has been used as the proxy/server platform and Jini as the service discovery protocol. It should be noted, however, that the µJini protocol API is operating system and programming language independent. For example, the client and proxy/server could be written using C++ on a Windows CE OS and Windows 2000 OS, respectively. In addition, the proxy/server can also be written to understand many different discovery protocols either instead of or in addition to the Jini discovery methods. New peer-to-peer (P2P) architectures will also be a viable means of service discovery for the proxy/server.

Thesis Organization

We begin in Chapter 2 with a survey of different service discovery systems available today. Chapter 3 covers different types of mobile devices on the market as well as their different operating systems or virtual machines that have software development kits available for use by developers. Chapter 4 describes the JiniDEN project by looking at the overall architecture, protocol API, and thin-client structure. Chapter 5 examines the different options for delivering the service to the user. Chapter 6 describes some

³ A J2ME compliant device provides a Java KVM (Kilobyte Virtual Machine) with supporting class libraries for the developers.
additional work done to the architecture: a proposed new protocol for appliances to provide an interface to users. Finally, chapter 7 concludes the thesis and looks at possible future directions in this area of research.
CHAPTER 2
SERVICE DISCOVERY TECHNOLOGIES

As computing and networking devices become smaller and cheaper, devices such as VCRs, TVs, home alarm systems, and refrigerators will all have small computers inside of them that are comparable to or have more processing power than the average desktop computer sold today. They will connect and disconnect from networks as needed for interaction with other network devices or as the network connection is established and lost. Jini, UPnP, and Salutation all support forms of distributed computing and have been around for several years. They each use discovery, lookup, and service invocation as the main protocols in their operations, although each individual system may define different formal names for the protocols. Two of the most recent ventures into this area are Sun’s Jxta P2P APIs [11] and Microsoft’s .NET technology [12]. Jxta and .NET are not necessarily service discovery techniques, but they can be used this way and so are included in this discussion. All of these systems are relatively new compared to the computing industry as a whole and there is still a lot of research, development, and testing on these products. Their use is mostly found in educational and high-tech research areas at this time. This is still far from the goal of the technology—to have these systems in common, everyday home appliances and devices.

When a device first encounters a network connection, whether by first attaching to the network or by the network suddenly recovering from a failure, it must notify other network devices that it is has joined the network. The device must also notify the
network of what services, if any, it is capable of providing to other services on the network. A device can be a client, a service provider, or both. Each protocol has its own way of doing this initial setup, although they are all similar in theory.

Devices on a network will often need to use services provided by another device. To do this, the device must initiate a lookup protocol to find out if the service it needs is available on the network. Services are provided in different ways in the different systems. Some will work by a proxy mechanism that only allows communication through a central device. Others allow direct connection with other devices.

There are several questions that must be addressed with respect to error conditions during execution of a service. What happens when something goes wrong with the network connection? What if a service fails, but the lookup service is not aware of the failure? What if a service suddenly fails during communication? Error situations can be handled using messages that represent remote exceptions in most cases. In cases where one entity experiences a crash or failure, a leasing protocol with timeout periods can be established. When a lease expires on either end of the connection, that entity should close its connection and clean up allocated resources.

**Jini**

Jini, by Sun Microsystems, was released as a final product in January 1999. Jini is based on Sun's Java language and uses RMI as a major part of its communication system. RMI allows the passing of not only data through the network, but code as well. Jini uses Java *marshalled objects* to pass object references across the network. A marshalled object will contain the data to represent an instance of an object as well as a location to download the object definition if not available locally. As long as a device
has a JVM installed or access to another network computer that can execute the Java code for them, it is relatively simple to set up a basic system since the Jini and Java APIs are encapsulating a lot of the basic leasing, underlying transport operations, and concurrency details. Jini should work with any network transfer protocol that has a JVM to support the network communications.

**Discovery**

When a Jini device first establishes a connection with a network, it will go through a process of discovery and join. In the discovery process, the device will look for a Jini lookup service on the network. If addresses are known for one or more lookup services, the device can attempt to connect directly to the lookup services. If there are no known lookup services, the device will dynamically find lookup services with a multicast on the network. If the client finds a lookup service, it will register with it so other devices on the network can use the services it provides. The device will upload a Java object to the lookup service for every service that it can provide to the network. This Java object will serve as an interface for the service to other Jini participants.

**Service Lookup**

Jini allows the action of service discovery by the way of a lookup service. When a device on the network wishes to use a service, it will make a request to a lookup service to see if the service or multiple services of the type needed are available. If the service exists on the network and is known about by the lookup service, a Java object with the service interface code encapsulated will be returned to the device that made the request. This object contains all the code and device-specific information required for the client device to use the service provided by the Jini device.
Service Use

A Jini service provides an object to a lookup service that implements a certain interface. A device on the network wishing to lookup and use this service must first have the interface available locally that defines the method calls available from the service. This allows the serialization of the object from the lookup service to the client device. Once the object has been recovered (instantiated) on the client, then method calls can be made and results returned. These method calls can be local or remote – the calling and return coding is the same for the client device and is transparent to the user. If calls are local, the code was returned in the service object and is executed on the client. If the calls are remote, the service object contains remote method calls and the code is executed on the service side and results returned to the client using RMI.

Error Handling

Jini provides a leasing system where each service obtains a lease from the lookup service for the period of time it will remain on the network. A service must periodically renew the lease or it will be removed from the lookup service. This ensures that if a service becomes unplugged from the network or fails, other devices on the network will not waste time trying to communicate with the failed device.

The Jini APIs define remote exceptions that can be thrown across the network when making remote method calls using RMI. The client-side code will use the java try-catch-finally syntax to catch remote exceptions and act upon them appropriately.

UPnP

UPnP, a joint venture by many prominent companies in the computing industry, was designed only in the past few years. It is an extension of the Plug and Play (PnP)
initiative started by Microsoft and other industry leaders in the early 1990s to facilitate ease of use in connecting desktop computers and peripherals. UPnP is based on existing open protocols such as HTTP, XML, DNS, and LDAP to pass messages and service information through the network. UPnP is not just for IP networks. It can be used with any type of network that supports the existing internet protocols mentioned above.

**Discovery**

UPnP devices can use a couple of different methods to announce themselves on the network: Automatic Private Internet Protocol Addressing (Apipa) and multicast name resolution. Apipa will be used where a Dynamic Host Configuration Protocol (DHCP) server does not exist. UPnP devices will assign themselves a 16-bit number and broadcast it over the network to see if there are any conflicts. Multicast name resolution extends the DNS (Domain Name Service) protocol to provide a DNS service in a small network. As the name implies, the name to address resolution is provided by multicasting a request and waiting for a response. In this case, each device in the network is responsible for the name to address resolution.

**Service Lookup**

UPnP uses Simple Service Discovery Protocol (SSDP) to lookup other services using a broadcast on the network. If a matching service is available, the device with the service will respond with the details of the service it will provide. This information is passed using a Uniform Resource Locator (URL) for the service device address and a block of Extensible Markup Language (XML) code to describe the service it provides.
Service Use

UPnP works by directly communicating with other devices. Once the service is discovered and the interface known, the client and service act as a normal client/server relationship. It is possible to insert a proxy server between the client and service to take care of smaller services not capable of being a full UPnP device.

Error Handling

UPnP does not explicitly check on any devices for availability. Since there is no central manager in UPnP, devices handle their own errors on lookup and failure. If a service is not available when a request broadcast is made, there will simply be no response to the request. If a failure happens during interaction between two devices, it is up to the devices to handle the error in their own manner.

Salutation

Salutation was started in July of 1995 and is the most mature of the five technologies. It also appears to be the most flexible which also makes it the most difficult to understand and implement. Salutation claims to be implemented more than any other of the technologies. If true, this is probably because it has been around the longest amount of time. Salutation works around Salutation Managers (SLMs). The SLMs may be on the device or they may just be accessible by the device. The SLMs handle all the communication between devices by talking with other SLMs. They keep track of where the other SLMs are and what services are at that location. If any data need to be transferred between devices or function calls made to and from devices, this is done through the SLMs with Remote Procedure Call (RPC) and External Data Representation.
Salutation can work in any networked environment that supports multiple and reliable bi-directional communications.

**Discovery**

Salutation devices perform all of their communication through the SLMs. Upon entering a network, the device must connect to an SLM if it does not have its own SLM built into it. The device will do this by Remote Procedure Calls. An SLM is not required to support devices by RPC. It is left to the individual implementation of the SLM.

**Service Lookup and Use**

Salutation devices will use their SLM to find other services on the network. Each SLM is capable of discovering other SLMs and the services they have available. If any data needs to be transferred or function calls made, those activities are done through the SLMs with RPC and XDR.

**Error Handling**

Salutation will take a similar approach to the Jini specification. Each SLM can do an availability check to see if a service is available. It does this with RPC calls to the other SLM that handles the Salutation device interaction.

**Jxta**

Jxta is a relatively newcomer to the computing scene (introduced in April 2001 by Sun Microsystems, Inc.). It is presented as a set of peer-to-peer (P2P) API specifications for developers to build P2P networks. Using a P2P network, it is very possible and probably imminent that this will develop into a dynamic service discovery system since
the very nature of a P2P network is to discover new peers and their services in a dynamic way.

Jxta messages are based on the Extensible Markup Language (XML) specification. These are text-based messages and can be parsed with nearly any device capable of basic computing services. The core protocols are shown in Table 1.

<table>
<thead>
<tr>
<th>PROTOCOL NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer Discovery Protocol (PDP)</td>
<td>The mechanism by which a peer will advertise its own resources and find the resources that other peers have to offer.</td>
</tr>
<tr>
<td>Peer Resolver Protocol (PRP)</td>
<td>The mechanism by which a peer can send queries to other peers after discovery by PDP.</td>
</tr>
<tr>
<td>Peer Information Protocol (PIP)</td>
<td>The mechanism by which a peer can obtain status information about other peers.</td>
</tr>
<tr>
<td>Peer Membership Protocol (PMP)</td>
<td>The mechanism by which peers will organize and group themselves. PMP messages allow peers to join and leave groups. All peers are members of the World Group.</td>
</tr>
<tr>
<td>Peer Binding Protocol (PBP)</td>
<td>The mechanism by which a peer can establish a virtual communication channel between two or more peers. Pipes are the fundamental communication between peers.</td>
</tr>
<tr>
<td>Peer Endpoint Protocol (PEP)</td>
<td>The mechanism by which a peer can discover a route to another peer. The route will be a sequence of hops through other peers.</td>
</tr>
</tbody>
</table>

Using these core protocols, peers can form groups and interact in many different ways. The Jxta community members are free to define new services and ways of interaction. The Jxta specifications only give a base layer on which P2P systems can be implemented.
Compact .NET

.NET is Microsoft’s architecture for creating web services using a language independent runtime environment and platform independent communication protocol. The .NET platform is still in beta testing at this time, but the full version should be released soon. The .NET Compact Framework is still in the specifications stage and this has not been released from Microsoft at this date. It is most likely that the .NET Compact Framework will allow small devices the capability to execute applications based on a small subset of the .NET Framework. In addition, small devices should be able to use .NET web services executing on more powerful server machines.

Overview

.NET is a significant project by Microsoft and consists of many components. The Common Language Runtime (CLR) is the main component of .NET and is the platform for which programs written for this platform will target. The CLR is somewhat comparable to Java’s JVM concept. .NET programs will execute within the boundaries of the CLR. The CLR will take care of class loading, security issues, and garbage collection. .NET supports a Common Language Infrastructure (CLI) so many languages can interact and use modules created in other languages. This is possible through the Common Language Specification (CLS). The CLS says that all compiled code should be in the Microsoft Intermediate Language (MSIL) format. This is a code+metadata file that describes the module. The Just-In-Time (JIT) compiler of the operating system will compile the MSIL code to native code for the operating system and computer hardware. .NET supports web services by using Universal Description, Delivery, and Integration
(UDDI), Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and XML technology in general.

**Service Registration, Lookup, and Use**

Although there are no formal specifications at this time on how service registration and discovery are to be performed, it would probably be similar to the layout shown in Figure 1. There will be some sort of web service registry or registries available to users on a network. The web service will use UDDI, WSDL, and SOAP to register with the Web Service Registry. Clients on the network wishing to use a web service will also use UDDI, WSDL, and SOAP to find services they need. Once a service has been located, the client may interact with the service by using SOAP.

![Figure 1: Probable .NET Service Discovery Scenario](image-url)
CHAPTER 3
MOBILE DEVICES AND DEVELOPMENT PLATFORMS

At this time, there are many new mobile computing devices coming to market in a short period of time. Processor speed, memory resources, and screen sizes all vary a great deal. This chapter lists just a few of the options that are available to consumers in today’s market.

Motorola i85s

The i85s [13] is the platform of choice for the µJini project. It was chosen because of the equipment provision by Motorola and because this is the first and only J2ME certified phone available in the United States at this time. We also received a provision of packet data service from Nextel that enabled us to use TCP and UDP socket connections on the phone. The phone has a routable IP address and can be reached through Nextel’s Mobile IP system. A picture of the i85s device is shown in Figure 2.

J2ME

The development platform available to third-party developers is a small JVM called Kilobyte Virtual Machine (KVM) with some additional class libraries that provide base functionality for the java language on the phone. The J2ME platform on the phone implements the CLDC 1.0 [14] and MIDP 1.0 [15] specifications. The CLDC specifications are a subset of the J2SE libraries and provide a rich java API for developers. The MIDP specifications are another layer on top of the KVM and provide
Figure 2: Motorola i85s Phone

LCD user interface libraries for working with a screen as well as some additional features that make the programmer’s job easier. A valid java program that can be executed on a J2ME device is called a MIDlet.

Memory

The i85s has three types of memory: data memory, program memory, and heap memory. The phones are supplied with 256kB of data memory, 384kB of program memory, and 256kB of heap memory. Heap memory is used for runtime execution of programs. Data memory is used for initial download storage of a MIDlet, persistent storage for MIDlet programs available through APIs, and voice recordings handled by the native phone OS. Program memory is used to store a program in an executable image format after it has been loaded from data memory, expanded, and verified from the data memory.
Persistent Storage

The MIDP specifications provide an API for persistent storage in the form of a RecordStore. This is a java object that can be instantiated and used to store data objects as raw bytes. It is the responsibility of the programmer to store and retrieve the information in the correct order.

Palm

The Palm Corporation licenses the Palm OS to companies such as Handspring for use on handheld devices. They also manufacturer their own handheld device with the Palm OS operating system installed. We will profile only the Palm m505 [16] model that is at the higher end of Palm’s product line at this time.

The m505 is shown in Figure 3 and has dimensions of 4.5” x 3.1” x 0.5.” The m505 contains a 33MHz Motorola Dragonball VZ processor with 8MB of RAM. It ships with the Palm OS 4.0 and some productivity software. Network connectivity can be established with the mobile internet kit that allows an internet connection through a serial modem or data-enabled mobile phone modem. The device has a 160x160 pixel color screen capable of displaying over 65,000 colors. There is an expansion slot that allows for add-on devices such as external storage or content devices. There is also a USB port for synchronizing data with a desktop computer or attaching a keyboard.

There is a native software development kit for the Palm as well as an implementation of J2ME that can be installed so java programs that meet the J2ME specifications can be executed on the device.
The iPAQ [17] is a small handheld device manufactured by Compaq Corporation. There are several versions on the market, but we will look at the 3650 model as shown in Figure 4.

The iPAQ uses a 206 MHz Intel StrongARM 32 bit RISC processor with 16 MB of ROM and 32MB of RAM. The screen is 240 x 320 pixels with a .24 mm pixel pitch, supporting 4096 colors and uses color reflective thin film transistor (TFT) LCD technology. There is a USB port as well as an infrared interface port. Extra persistent storage space can be added through expansion modules such as an IBM Microdrive card. Network connectivity can be added through a PCMCIA slot attachment with an 802.11 card or Bluetooth card inserted.

The iPAQ ships with the Windows Pocket PC operating system by Microsoft Corporation, although it can have other operating systems installed such as Linux. The Pocket PC operating system is a smaller version of the near-ubiquitous Windows platform developed by Microsoft. Microsoft supplies many of the familiar Windows
programs to use on the platform as well as a free development environment for third-party providers to develop applications for the Pocket PC platform.

**NTT DoCoMo and iMode**

NTT DoCoMo [18] is a company based in Japan that provides a service called iMode. This is a wireless service technology for delivery of voice and data services to devices. An iMode device can make voice calls as well as use packet data service. The iMode service uses Compact HTML (CHTML) whereas Wireless Application Protocol (WAP) uses Wireless Markup Language (WML). The newer iMode phones such as the one shown in Figure 5 also support Java applications by using a KVM with a CLDC profile installed on the device similar to Motorola’s i85s phone described in an earlier section of this chapter. The iMode service now also includes a feature called “i area.” This is a location-based feature that lets a user retrieve content relative the area surrounding the location of the phone.
The low-level operating system and development platform is proprietary, but the KVM+CLDC layer allows for a java application space on the phone so third-party application developers can write add-on programs for the device.
In this chapter, the µJini architecture is explained in detail. As noted before, Jini does not normally fit on the small devices that implement J2ME with CLDC and MIDP because of the limited memory size, persistent storage size, processor speed, and limited support libraries; hence the need for an architecture for smaller devices that will allow service discovery and delivery similar to the Jini scenario. In addition, with our architecture we strive to provide support for not only Jini services, but any other type of service our proxy can support. Since the proxy can be any machine on any platform, we can support nearly any service available such as Salutation, .NET, or a Bluetooth profile.

The µJini architecture has been developed with other University of Florida graduate students Choonhwa Lee [19] and Sam Coons [20] over the period of approximately one year.

**Overall Layout of µJini**

Figure 6 shows the main pieces of the µJini architecture: the protocol, the proxy, the client, and the communication channels.

The client is a small, networked J2ME device with low resources. The proxy is a J2SE application running on a networked computer with virtually unlimited resources available. The µJini protocol on top of TCP/IP is how the client and server communicate. The TCP/IP layer is used for the socket communication between the client and the Virtual Network Computing (VNC) [21] server.
Protocol

The protocol is how the client and proxy server communicate. Java 2 supports serialization of objects for message-passing, but J2ME is built for low-resource devices and neither serialization nor RMI is supported on this platform. To implement message-passing between client and server in the µJini system, we have created a set of protocol classes that allows messages to be sent and received across the network using a simple API that is consistent on the client and server sides. Both of the applications use the same set of java API classes, with the only difference being the client side codes.
must be preverified\(^1\) before installing onto the J2ME client device. The classes have been written using Java classes available to the CLDC 1.0 with the MIDP 1.0 classes as well as the J2SE libraries. Since this library is shared between two different sets of APIs, care must be taken to write the libraries with support for both APIs while also making them efficient, useful, and intuitive to the developer. Because of the low resources available on the client, the libraries have also been written to occupy the smallest storage and execution space possible. Common object-oriented programming techniques such as information-hiding and get/set methods have been eliminated in many cases to reduce the space needed for method description in the class files. Variables are simple and are allowed public access from other classes.

There is one base class named \textit{MJBaseProtocol} from which all other message objects are derived. This class also contains a static method named \textit{createFrom} that is used to construct all µJini messages from an input stream. After construction of a message from a stream, the resulting object’s message type can be found with the Java \textit{instanceof} operator. All message types have constructors for creating the objects. All messages also have a \textit{sendTo} method for sending a message on a given data stream. This method takes an output stream as a parameter and writes all the bytes to the stream in the most efficient manner (as a block of data). With this technique we are attempting to send as few possible network packets as possible. The performance of wired networks is typically very fast and the performance degradation is not seen clearly as it is on our project platform of a cellular wireless network using Mobile IP.

\(^1\) The preverification process is a requirement of the MIDP specifications. Preverification adds some additional information to the compiled class files for security checks.
Table 2 shows the message types and their intended use in the communication between client and proxy within the µJini system.

**Client**

The client is named *ResidentClient* and is executing on a J2ME device that supports the CLDC 1.0 and MIDP 1.0 libraries. The *ResidentClient* is a MIDlet and runs as a normal J2ME MIDlet on the i85s phone. An intelligent option by a device manufacturer would be to provide the *ResidentClient* as part of the firmware code so it is not taking application space that can be used for additional user applications. The *ResidentClient* provides a graphical user interface to the user for using the µJini protocol to find and request services from the proxy server. It will also take care of instantiating a thin-client viewer (the VTC) and cleaning up resources after a service session is complete. In the case of downloaded services, it will interact with the proxy, download the MIDlet from an http server, and load the code onto the phone using native operating system functions.

**Discovery**

The client will send an *MJProxyDiscoveryRequest* message onto the network to discover a new proxy within the local network area. The client will then send an *MJClientConnectRequest* message to a particular proxy once a proxy server address is known. For security or convenience, a list of known proxy servers may be kept on the device for use at any time.

The proxy will respond to the *MJClientConnectRequest* message with either an *MJAcceptClient* to accept the client connection or an *MJDenyClient* message if the proxy is not allowing the client to connect.
<table>
<thead>
<tr>
<th>Message Class Name</th>
<th>Message Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>MJAcceptClient</td>
<td>Proxy returns to client after MJClientConnectRequest</td>
</tr>
<tr>
<td>MJBaseProtocol</td>
<td>None</td>
</tr>
<tr>
<td>MJClientConnectRequest</td>
<td>Client initial message when connecting to a proxy</td>
</tr>
<tr>
<td>MJClientDisconnecting</td>
<td>Client sends to proxy when finished with session</td>
</tr>
<tr>
<td>MJClientDiscoveryResponse</td>
<td>Client sends to a proxy after a dynamic proxy discovery message was returned</td>
</tr>
<tr>
<td>MJClientException</td>
<td>Client sends to proxy if client error has occurred</td>
</tr>
<tr>
<td>MJDenyClient</td>
<td>Proxy returns to client after MJClientConnectRequest</td>
</tr>
<tr>
<td>MJEntry</td>
<td>An entry to be used inside an MJServiceLookup message as an attribute of some type of service to be discovered</td>
</tr>
<tr>
<td>MJKvmEvent</td>
<td>An event from the client to the proxy when in VTC mode and the service is a MIDlet being run on an emulator within the proxy domain</td>
</tr>
<tr>
<td>MJLeaseRenewal</td>
<td>A Client message to the proxy to retain leasing rights and not be disconnected</td>
</tr>
<tr>
<td>MJmsg</td>
<td>A class to retain static constants in use throughout the protocol library</td>
</tr>
<tr>
<td>MJOBJECT</td>
<td>A base object that can be used to represent different data types that might be serialized across the network within the protocol library</td>
</tr>
<tr>
<td>MJProxyAnnouncement</td>
<td>A message the proxy multicasts on a network to let clients and other proxies know it exists</td>
</tr>
<tr>
<td>MJProxyDisconnecting</td>
<td>A proxy will send this to a client when it initiates a disconnect</td>
</tr>
<tr>
<td>MJProxyDiscoveryRequest</td>
<td>A client multicasts this message on a network when it is discovering new proxies</td>
</tr>
<tr>
<td>MJProxyDiscoveryResponse</td>
<td>Proxy sends to client after a dynamic proxy discovery message has been received</td>
</tr>
<tr>
<td>MJProxyException</td>
<td>Proxy sends to client if a proxy error has occurred</td>
</tr>
<tr>
<td>MJRequestVTC</td>
<td>Client sends to proxy after discovering a service and wanting to obtain a VTC reference to execute</td>
</tr>
<tr>
<td>MJRequestVTCResponse</td>
<td>Proxy sends to client in response to a MJRequestVTC message</td>
</tr>
<tr>
<td>MJServiceDisconnecting</td>
<td>Client sends to proxy when finished with a VTC session</td>
</tr>
<tr>
<td>MJServiceLookup</td>
<td>Client sends to proxy when requesting a certain type of service reference</td>
</tr>
<tr>
<td>MJServiceLookupResponse</td>
<td>Proxy sends to client in response to a MJServiceLookup message</td>
</tr>
</tbody>
</table>
Service Lookup

The client will send an MJServiceLookup message to the proxy to discover a certain type of service. The MJServiceLookup message contains many important pieces of information such as the current device resources that are available, the name of the service for which to search, and a list of attributes the service may have. We have written our proxy server to search as in a Jini search where we search by an interface class name or by Entry classes that serve as attributes. Attributes we have defined are ones such as City, Building, Room, and Street. If a proxy were to support other service discovery architectures then those service names and attributes could be matched in a similar manner. The proxy will return a message of type MJServiceLookupResponse with the number of services found.

Using the Service

For each service found, there are attributes for the name of the service, the type of service (MIDlet or Standard), whether the service can be downloaded to the device if a MIDlet, and the number of parameters a service will take if it is a standard type service. In the case of a standard Jini service, the number of parameters will be accompanied by an array of types for the parameters (primitive types such as int, float, string, etc.) as well as a parameter field name. The client will then dynamically construct a form for the user to fill out and send the user supplied parameters to the proxy with which to execute the service. The client may decide to download the application if the proxy sets a flag to indicate it is ok. If not the client must decide to use the thin-client approach or not use the service at all.
**Proxy/Server**

The proxy is written using the J2SE platform and is much more powerful than the client. It is expected that the proxy will run on a powerful desktop machine or server. It houses the VNC server used to run applications for the client while in the thin-client mode. The MIDP emulator used to emulate MIDlets for thin-client mode runs on top of a VNC server session. The VNC server provides a virtual windowing environment for GUI applications. A proxy should announce itself on the network periodically by sending an `MJProxyAnnouncement` messages by multicast to a well-known port. The proxy should also listen on a well-known port for `MJProxyDiscoveryRequest` messages being multicast on the local network by client devices or other proxies.

**Service Discovery**

The JiniDEN implementation uses Jini for discovery and can discover all lookup services and registered services on the local area network as well as any hard-coded lookup services that may be added for external networks. The proxy accepts class name and attribute parameters from the client using the µJini protocol. It then maps those into a syntactically correct Jini lookup message. A list of all discovered services is then sent back to the client using the µJini protocol.

**Service Hosting**

The proxy will use a VNC server running on a unix platform to execute applications the client wishes to run in thin-client mode. A Windows machine could be used, but at this time there is only support for one screen on the Windows platform. We desire to have the ability to create many virtual desktops to be used among many different clients. When the proxy starts, it will allocate a certain number of VNC
desktops that will be available when needed. More can be started later if needed, but requires additional service startup time that should be kept to a minimum. For a standard service type such as an xterm application or map service through a web browser, the proxy can simply start up these unix programs on a reserved VNC desktop that is already running. For a MIDlet service, the jad and jar file must be downloaded via http, the files unpacked and verified, and then the MIDlet executed using a device emulator running on top of one of the VNC servers.

**Message Passing Example**

As an example of how a session might proceed between client and server, see Figure 7. The client code for the scenario is shown in Appendix A. The code is a type of java pseudo-code and is only a method contained within a complete program. Some method implementations are not shown, but should be self-documenting as to the function they perform. Note that little error handling is done here and that some threading should be done to capture datagram packets in the background while the program still continues. In this example the client is searching for a map service to obtain a map of the current surroundings.

The proxy/server code for the Figure 7 scenario is shown in Appendix B. The code is a type of java pseudo-code and is only a method contained within a complete program. Some method implementations are not shown, but should be self-documenting as to the function they perform. Note that little error handling is done in the example code.
Figure 7: µJini Communication Example
CHAPTER 5
SERVICE DELIVERY IN MICROJINI

Service delivery in the µJini system will occur as a downloadable application that will be executed locally on the client device or as an application that runs on a remote server in a thin-client mode. Downloading an application to the phone over the airwaves can take quite a bit of time. An application that is only 34kB in size will normally take around 43 seconds to download to the phone with the wireless network speed we are using for our project. After downloading, the application still has to be installed onto the device in a verification and program image-creation process that takes an average of 190 seconds for the same 34kB application. This results in the user waiting for nearly four minutes to use the application that has been discovered. In a mobile environment, it is quite likely the user has moved to another location after this length of time and is no longer interested in the application discovered four minutes ago. There must be a faster way to get these services onto the user’s device. In this chapter we discuss the VTC method of accessing the service. VTC is based on thin-client technology so we must first give a brief introduction to VNC that our system is based upon. Next we look at the methods used in µJini to deliver a service and optimizations used to improve overall performance of the VTC model within the µJini implementation.

Overview of VNC

VNC is a thin-client system. A thin-client system is one where the program executes on a remote server, while the processing done by the local client is minimal.
The client’s responsibility is to send events, such as keyboard and mouse events, to the server for processing and receive back screen changes that have occurred within the application on a periodic basis. The client that conforms to these specifications can be a very limited device, resulting in a much cheaper and/or smaller computing device than would normally be possible to execute a complex application. This situation is ideal for the new mobile computing devices that are typically resource-constrained devices and need to run larger applications on a remote server such as a database or spreadsheet application.

**RFB Protocol**

RFB is an acronym that represents the term *Remote Frame Buffer*. The VNC system revolves around the RFB protocol and its definition. The server maintains the state of the application at all times in a data structure called the framebuffer that represents the application display. Clients can connect, disconnect, and reconnect continually with no ill effects on the server – the state will be the same as the last client has left it. If the server allows, many clients can be connected at once and all interacting with the server application in parallel. Updates to the client are made with reference to rectangles. Each update will be to place a rectangle of a certain size with a certain color at a certain location. Optionally, an update may consist of any number of rectangle updates combined into one large update as an array of rectangle updates. The updates are mandated by the client. The server will only send an update to the client if a message has been received by the server that the client is requesting an update. The server will keep a list of updates that a client needs and when a timeout period occurs (40ms default) and a client update request is pending, then the server will send the update to the client. The
client can request incremental updates or full updates on any area of the framebuffer that the client requires. If the client requests incremental updates, then only updates since the last update will be sent. If the client needs all framebuffer data, a non-incremental update can be requested for an area and all the data will be sent. This could be useful for a device that has a high-speed network connection, but low memory resources to store the image data locally.

The initial RFB encodings defined by AT&T for rectangle updates are shown in Table 3.

<table>
<thead>
<tr>
<th>ENCODING NAME</th>
<th>DESCRIPTIVE NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>Send each pixel in framebuffer from top row to bottom row and each row left to right</td>
</tr>
<tr>
<td>Copy Rectangle</td>
<td>Copy of section of the screen another location</td>
</tr>
<tr>
<td>RRE</td>
<td>Rise-and-run-length, a two-dimensional equivalent of run-length encoding</td>
</tr>
<tr>
<td>CoRRE</td>
<td>Compact rise-and-run-length, a variant of RRE where each rectangle is no larger than 255x255 pixels</td>
</tr>
<tr>
<td>Hextile</td>
<td>A variation of the CoRRE encoding idea, where each rectangle is 16x16 pixels except for end of row or column rectangles which are allowed to be smaller to fit the image width and height</td>
</tr>
</tbody>
</table>

**Handshaking**

The VNC implementations available today can have initial connections made by the server or client. In our case we will only look at the traditional client/server relationship where the server waits for a client connection, conducts the initial handshaking protocol, and then proceeds with processing the normal RFB protocol messages. The handshaking proceeds with the server sending a ProtocolVersion
message. This is a twelve-byte string in the format “RFB xxx.yyy\n” where xxx defines
the major number for the RFB protocol and the yyy defines the minor version of the RFB
protocol supported by the server. The client will then send a message in the same format
that tells the server what the latest RFB protocol version it can support. With this
information it is possible for the server and client to provide backwards compatibility
with older clients and servers. The server will then send an Authentication message to
the client. This message can either be that the connection has failed, there is no
authentication needed, or that VNC authentication is needed. For no authentication the
client will simply send the ClientInitialization message. For VNC authorization, the
server also sends a 16-byte challenge. The client is expected to use DES to encrypt the
challenge using a password supplied by the user, and return a 16-byte response to the
server. The server will then reply whether the challenge-response was successful. If it
was not successful, the server then closes the connection. If it was successful, the client
then sends the ClientInitialization message. The ClientInitialization message contains a
flag saying whether the client is willing to share this connection with other clients or
whether the server should disconnect other clients from the VNC session. The server
then sends the ServerInitialization message. This message tells the client the framebuffer
width, height, pixel format, and desktop name. The pixel format contains the information
for bits per pixel, the depth of each bit, a big-endian boolean flag, a true-color flag, a red
maximum and shift value, a green maximum and shift value, and a blue maximum and
shift value. At this point the handshaking is complete. The client can now send
additional messages to the server and the server can send messages to the client.
Client Messages

The client may send a number of different messages to the server at any time. These messages are described in Table 4.

Table 4: RFB Protocol Client Messages

<table>
<thead>
<tr>
<th>MESSAGE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetPixelFormat</td>
<td>Tell the server to send pixel data in a certain format. The pixel format contains the information for bits per pixel, the depth of each bit, a big-endian boolean flag, a true-color flag, a red maximum and shift value, a green maximum and shift value, and a blue maximum and shift value. This information overrides the format set by the server in the ServerInitialization message.</td>
</tr>
<tr>
<td>FixColourMapEntries</td>
<td>There is little or no support on current VNC implementations for this message. This is meant so the client can tell the server how it wants certain pixel values mapped to certain colors.</td>
</tr>
<tr>
<td>SetEncodings</td>
<td>Tell the server what types of encoding the client can understand. The order of the list of encodings also tells the server our order of preferred encodings.</td>
</tr>
<tr>
<td>FrameBufferUpdateRequest</td>
<td>Tell the server to send an update of the framebuffer. The request provides the area of the framebuffer to be sent and also a flag to say whether the client wants the entire area sent unconditionally or if the client only wants updates since the last framebuffer update was sent.</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>Tell the server that a key event has occurred on the client side. Information is sent describing the type of key and whether this event was a key press or a key release.</td>
</tr>
<tr>
<td>PointerEvent</td>
<td>Tell the server that a pointer event has occurred on the client side. Information is sent describing the location of the pointer in an x-y coordinate system and the state of up to eight buttons (either up or down).</td>
</tr>
<tr>
<td>ClientCutText</td>
<td>This is an announcement that the client has new ASCII text in its cut buffer. The text is attached to the end of the message.</td>
</tr>
</tbody>
</table>
Server Messages

The server has several messages that it may send to a client. The

*FrameBufferUpdate* must only be sent if the client has requested an update with the

*FrameBufferUpdateRequest* message. The other messages may be sent to the client from the server at any time. The server messages are listed in Table 5.

<table>
<thead>
<tr>
<th>MESSAGE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FrameBufferUpdate</td>
<td>Contains an update of rectangles for the client to update its display.</td>
</tr>
<tr>
<td>SetColourMapEntries</td>
<td>There is little or no support on current VNC implementations for this message. This message is meant so the server can tell the client about color map entries the server wishes to set and for which the client has not already specified.</td>
</tr>
<tr>
<td>Bell</td>
<td>This is an announcement that a bell event has occurred on the server desktop.</td>
</tr>
<tr>
<td>ServerCutText</td>
<td>This is an announcement that the server has new ASCII text in its cut buffer. The text is attached to the end of the message.</td>
</tr>
</tbody>
</table>

Service Delivery

There are two types of service delivery that were considered for use in the μJini system: Standalone Applications or Virtual Thin Client (VTC). Some choices have to be considered when making the decision to download a service or to execute by thin-client. Sometimes the choice will be made by the server and other times the client will have a choice of either option. In this section we look at the many factors that must be considered.
Downloading

For the client to be able to download the service, the service provider must allow this action and the client device must also be capable of executing the service locally. When downloading the service, an http connection is made to the address supplied by the proxy so the MIDlet’s jad and jar file can be downloaded to the device. If the device allows software installation, the ResidentClient can initiate this after download. If not, the user must return to the application menu and install the downloaded service manually before it can be used. Since the service is downloaded to the client and executed locally, the client is free to leave the local area and still execute the program. A disadvantage is the time spent to download and install the program as well as the space taken by the program on the storage area of the device.

Virtual Thin Client

If the choice is made for VTC mode, the ResidentClient will send an MJRequestVTC message with the service parameters for the service the user has selected. The proxy will return an MJRequestVTCResponse message with a response that either it has successfully started the service or that the service cannot be started for some reason. If the service could be started, the connection information for the VNC server will be retuned within the MJRequestVTCResponse message. For the VTC mode, the application is run on the proxy’s VNC server and only screen updates are sent to the client. The client in turn only sends keyboard and/or mouse events to the proxy. It is the responsibility of the proxy to deliver these events properly to the MIDlet application. This requires a thin-client application to be present on the client at all times, which is named VTC.
The VTC is a scaled-down version of the java VNC client available from AT&T Research at Cambridge. The java client has been modified to execute as an application under the J2ME libraries. Most of the changes revolved around the screen manipulation routines that update the display upon receiving a framebuffer update message from the VNC server. The other major portion of the java client modification is in the key-handling section of the client. Since the phone does not have a keyboard, a mapping has to be made between the ITU-9 keypad of the phone and key events that would normally occur on a computer keyboard. The ITU-9 keypad has a sequence of alphanumeric characters associated with each key. For example, the key used for the number 2 is also used for the letters a, b, and c. Figure 8 Shows the keypad from the emulator screen for the i85s phone. For this project we chose to write a set of key-handling classes that wait

Figure 8: i85s Keypad Layout
for a timeout period between keypresses to determine the character the user wishes to send. If a key is pressed once and the timeout period expires or another key is pressed, then the first character of the key character array is sent. If the key is pressed more than once, the key character is rotated through the array of keypad characters until the timeout expires or another key is pressed. It is at this time that the key event is sent to the server. Sam Coons completed most of this work during the project and his thesis work [20] contains in-depth information about the implementation.

Because there is no requirement of download and installation of the program on the client, time can be saved in the VTC scenario. The amount of it takes to start up the service and use it are much less compared to the method of downloading a service and executing it locally. One drawback to VTC mode is the low bandwidth of current wireless networks and the resulting transmission times of key events and screen updates. The result is the application appears to be sluggish to the user. The other disadvantage of VTC over downloaded applications is that if the user loses contact with the proxy or VNC server, the service session will end and the user will not have use of the service any longer.

The Decision Process

After service discovery, the choice must be made whether to download the application to the device or to use a VTC to view and control the application running on a server machine. A choice can be made by the service provider to allow the downloading of the service to the client or to force VTC usage and keep the code private and on the server. If the service provider only allows VTC use, then the choice has been made for the client. If the service is downloadable, the client must make the choice of whether to
download the service or use the VTC method. This choice should be made software configurable and the user can have the choice made automatically or the user may wish to be prompted each time for a decision on the way the service is used.

**Performance for Delivery Options**

There is huge difference in the time between a downloaded service and VTC service usage. A service application that is run locally must first be downloaded to the client device and then installed before it can be executed. The low bandwidth wireless networks that are common today along with the long process of installing an application on a resource-constrained device add up to a long time for the user to wait to use the service. A service application that is run as a thin-client, however, only needs to wait for the server to start the service and supply the client with the VNC server address and port number with which to create a connection to the VNC server. The connection to the VNC server is very fast and the application can be started in seconds rather than minutes for a downloaded service. Table 6, Table 7, Table 8, and Table 9 show the results of tests in the lab for starting a VTC session versus downloading the jad and jar files, installing the application, and local execution startup time. The times varied greatly for each case. The data for VTC startup and downloading over many iterations reflects the wireless network performance instability. The install and startup time differences can be attributed to the fact that the phone is not a single-purpose machine and the native phone operating system has many tasks in the background to take care of. It is also the case that the phone may occasionally drop into a low-power mode when no activity has been noticed for a period of time. This is to conserve the battery life of the device.
Table 6: Performance Data for *iCoffee* MIDlet

<table>
<thead>
<tr>
<th>jad size(bytes)</th>
<th>jar size (bytes)</th>
<th>D/L (s)</th>
<th>Install (s)</th>
<th>Local Startup (s)</th>
<th>Total Local Startup(s)</th>
<th>VTC startup (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>373</td>
<td>35700</td>
<td>69</td>
<td>163</td>
<td>7</td>
<td>239</td>
<td>15</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>49</td>
<td>147</td>
<td>7</td>
<td>203</td>
<td>11</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>57</td>
<td>180</td>
<td>5</td>
<td>242</td>
<td>9</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>87</td>
<td>183</td>
<td>5</td>
<td>275</td>
<td>10</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>52</td>
<td>159</td>
<td>5</td>
<td>216</td>
<td>13</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>37</td>
<td>177</td>
<td>5</td>
<td>219</td>
<td>13</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>58</td>
<td>185</td>
<td>5</td>
<td>248</td>
<td>16</td>
</tr>
<tr>
<td>373</td>
<td>35700</td>
<td>54</td>
<td>195</td>
<td>5</td>
<td>254</td>
<td>10</td>
</tr>
<tr>
<td><strong>averages</strong></td>
<td></td>
<td>58</td>
<td>168</td>
<td>6</td>
<td>232</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 7: Performance Data for *Shopping* MIDlet

<table>
<thead>
<tr>
<th>jad size(bytes)</th>
<th>jar size (bytes)</th>
<th>D/L (s)</th>
<th>Install (s)</th>
<th>Local Startup (s)</th>
<th>Total Local Startup(s)</th>
<th>VTC startup (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>339</td>
<td>23074</td>
<td>54</td>
<td>93</td>
<td>7</td>
<td>154</td>
<td>10</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>58</td>
<td>99</td>
<td>7</td>
<td>164</td>
<td>9</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>37</td>
<td>85</td>
<td>7</td>
<td>129</td>
<td>10</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>37</td>
<td>90</td>
<td>4</td>
<td>131</td>
<td>13</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>52</td>
<td>92</td>
<td>3</td>
<td>147</td>
<td>10</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>37</td>
<td>90</td>
<td>3</td>
<td>130</td>
<td>13</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>46</td>
<td>92</td>
<td>4</td>
<td>142</td>
<td>10</td>
</tr>
<tr>
<td>339</td>
<td>23074</td>
<td>43</td>
<td>82</td>
<td>7</td>
<td>132</td>
<td>11</td>
</tr>
<tr>
<td><strong>averages</strong></td>
<td></td>
<td>46</td>
<td>91</td>
<td>5</td>
<td>142</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 8: Performance Data for *PhoneClient* MIDlet

<table>
<thead>
<tr>
<th>jad size(bytes)</th>
<th>jar size (bytes)</th>
<th>D/L (s)</th>
<th>Install (s)</th>
<th>Local Startup (s)</th>
<th>Total Local Startup(s)</th>
<th>VTC startup (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>345</td>
<td>24010</td>
<td>37</td>
<td>98</td>
<td>2</td>
<td>137</td>
<td>10</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>43</td>
<td>102</td>
<td>2</td>
<td>147</td>
<td>13</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>38</td>
<td>103</td>
<td>1</td>
<td>142</td>
<td>10</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>36</td>
<td>101</td>
<td>1</td>
<td>138</td>
<td>13</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>36</td>
<td>105</td>
<td>2</td>
<td>143</td>
<td>10</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>49</td>
<td>113</td>
<td>1</td>
<td>163</td>
<td>10</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>40</td>
<td>111</td>
<td>1</td>
<td>152</td>
<td>9</td>
</tr>
<tr>
<td>345</td>
<td>24010</td>
<td>37</td>
<td>111</td>
<td>2</td>
<td>150</td>
<td>13</td>
</tr>
<tr>
<td><strong>averages</strong></td>
<td></td>
<td>40</td>
<td>104</td>
<td>1</td>
<td>145</td>
<td>11</td>
</tr>
</tbody>
</table>
Table 9: Performance Data for *MapIt* MIDlet

<table>
<thead>
<tr>
<th>jad size (bytes)</th>
<th>jar size (bytes)</th>
<th>D/L (s)</th>
<th>Install (s)</th>
<th>Local Startup (s)</th>
<th>Total Local Startup (s)</th>
<th>VTC startup (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>329</td>
<td>34777</td>
<td>40</td>
<td>188</td>
<td>8</td>
<td>236</td>
<td>16</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>42</td>
<td>187</td>
<td>8</td>
<td>237</td>
<td>10</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>42</td>
<td>186</td>
<td>7</td>
<td>235</td>
<td>10</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>38</td>
<td>178</td>
<td>7</td>
<td>223</td>
<td>15</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>37</td>
<td>201</td>
<td>8</td>
<td>246</td>
<td>9</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>57</td>
<td>200</td>
<td>7</td>
<td>264</td>
<td>14</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>37</td>
<td>193</td>
<td>7</td>
<td>237</td>
<td>10</td>
</tr>
<tr>
<td>329</td>
<td>34777</td>
<td>50</td>
<td>195</td>
<td>7</td>
<td>252</td>
<td>10</td>
</tr>
<tr>
<td>averages</td>
<td></td>
<td>43</td>
<td>190</td>
<td>7</td>
<td>240</td>
<td>12</td>
</tr>
</tbody>
</table>

In Figure 9 the performance data averages are shown for the total time to start the service after discovery. The comparison shows the large difference in time for a user to start a service when downloading compared to when using the VTC method.

![Comparison of Local Execution Startup and VTC Startup](comparison.png)

Figure 9: Service Startup Time Comparison
VNC Optimizations

VNC was designed for the client to be as dumb as possible. By this, it is meant that the client should not need a lot of processing power to receive the screen update data, process the data, and update the client display. The client does require, however, a significant amount of bandwidth sometimes to receive the screen data in a reasonable amount of time so the device and application do not appear sluggish. VNC supports the inefficient method of raw encoding of the pixels plus many other encoding schemes that are very efficient.

Because VNC is optimized for a dumb client, high-bandwidth use is often the trade-off. The VNC protocol typically works well on a wired Ethernet connection where network speeds are usually 10 or 100Mbps. There is a noticeable difference on a small device using a low-bandwidth wireless connection. To alleviate some of the delays we have implemented the new PNG protocol for VNC that allows faster updates for some types of screen images. We have also implemented a differencing algorithm on the VNC server side that makes an attempt to discover what area has changed pixel-by-pixel in the image and only ship that rectangle area as a framebuffer update.

VNC PNG Protocol

Since our development device, the Motorola i85s, uses PNG as its native image format, it seemed natural to adopt this for our image optimization protocol because our device would be able to decode and draw the image very quickly with minimal additional coding within the µJini client. Other reasons for choosing PNG are because the image format uses lossless compression and the PNG format has an open specification with plenty of source code in the public domain.
We first implemented the 8-bit color PNG image protocol. This creates a PNG color image file from the region of the screen to be updated. This framebuffer update sends the size of the image, the location on the display of the image, the length of the image byte stream, and the byte stream that contains the image representation.

The next protocol implemented was the 8-bit grayscale PNG image protocol. This creates an 8-bit PNG grayscale image file from the region of the screen that is being sent as a framebuffer update to the client and now sends instead the size of the image, the location on the display, the length of the image byte stream, and the byte stream that represents the image. In this case the server will grayscale the image when creating a PNG image using functions in the PNG library. There will be a reduction in the resulting image size for certain images that can be compressed further because of the resulting grayscale changes.

**VNC PNG Reduced-Bit Protocol**

In addition to using the new PNG image protocol for VNC, we can also add some reduced-bit protocols for limited capability clients that do not have color screens or only have a certain number of grayscale bits. By having the server reduce the image to a lower number of bits, the image may compress at a greater ratio and the resulting network transmission time and decode time on the client side will be reduced. Note that we always assume the processing power of the servers to be several magnitudes greater than the client. We chose to implement the 2-bit grayscale PNG image protocol. This creates a 2-bit PNG grayscale image file from the region of the screen that is being sent as a framebuffer update to the client and now sends instead the size of the image, the location of the image on the display, the length of the PNG image byte stream, and the byte
stream that represents the image. The server will iterate through the image pixels one by one and reduce each pixel to a 2-bit value according to an algorithm we have defined. Each pixel is still represented by 8 bits, but the resulting PNG image size will be smaller for images that contain many colors because the compression level will be increased due to the fewer number of colors in the image to represent. Note that there will be a loss of precision within the image, but if the client device only has 2-bit grayscale capability, then this will occur on the client side anyway. Since the server is more powerful and the image size can be reduced, it is more efficient in computing time and transmission time to do the image conversion on the server side before the image update is sent to the client.

VNC Framebuffer Differences

In addition to the PNG image format, it was noticed that the VNC server could not distinguish between a new image being drawn on the desktop or the same image drawn again in the same area of the screen. For example, many programs such as a Java application with a canvas will simply redraw the entire window again during a repaint system call instead of trying to determine what has changed and simply draw those parts of the screen. The MIDlet emulator program being used for this project is one such application that behaves this way if the MIDlet is redrawing the screen in this way. The VNC server receives these painting messages as the windowing system drawing calls and simply decides the client needs these new updates even if the exact same thing is drawn on the client side already. The new VNC server modified for this project creates a copy of the desktop framebuffer for every client. Before sending an update to a client, these buffers are now compared pixel by pixel to determine what the difference is. Only the
difference between the two buffers is sent to the client and the client buffer copy on the VNC server is updated to reflect the new desktop framebuffer.

**VTC Performance Measurements**

During this project we have implemented an 8-bit color PNG protocol, an 8-bit grayscale PNG protocol, and a 2-bit grayscale PNG protocol to complement the existing RFB protocols. These new protocols along with the differencing algorithm enhance the overall performance of the µJini system when the VTC option is invoked as the service delivery option. This section examines some basic network latency tests performed with the i85s phone using TCP and UDP packet data. Next we show some basic network performance notes with a relationship to the project implementation. The following sections show some performance tests on the system using the VNC server and emulator on the proxy side (test cases 1 and 2). Test cases 3, 4, and 5 show some specially developed tests using a server that allows more control of the screen updates. The tests refer to the phone emulator screen captures shown in Figure 10. The Diagonal_1, Diagonal_2, and Oval screens are shown as the original color images from the server in Figure 11.

**Network Latency Notes**

Wireless networks are known to have a much higher latency and bit error rate than wired networks. During testing of the protocol, it was noticed that it was taking a long time to get an entire message from client to server and vice-versa. For the initial coding, the protocol was written out as a sequence of bytes, a short, an int, or a long. After snooping the network traffic, it was realized that a packet was being sent for each
four bytes or more that were accumulated. Because the message data is fairly small, this might have gone unnoticed on a fast network, but was very obvious that something was

Figure 10: Phone Screens Used in Performance Tests

Figure 11: Original Test Images as Seen on Server
wrong on a connection that is wireless and supports a max speed of approximately 19kbps. A buffered writer class is not supplied in CLDC 1.0 or MIDP 1.0 so the μJini protocol libraries were rewritten to write each class message data to a byte array first and then send the entire byte array in one transmission. This greatly improved the speed of the message-passing between the client and the proxy server.

Even with the improvements stated in the previous paragraph, the network latency is still enough to cause very noticeable delays to the user. A few tests were designed and executed to show the average network latency using TCP communications that were encountered in this project. In addition we have also run some tests on UDP packets for comparison. In each test the client sends a packet of four bytes to the server and then waits for a return packet of a certain length. The server simply accepts the socket connection, reads in the first four bytes, and then returns a packet of a determined size. The server-side delay should be very minimal. We have run tests for the return data sizes of 4 bytes, 500 bytes, and 1000 bytes. These tests mimic the case where the client would send a few bytes to the server to indicate a keystroke and the server responding with a new screen update. The average test results are shown in Table 10.

### Table 10: Average Network Round-Trip Times

<table>
<thead>
<tr>
<th>Send Size (bytes)</th>
<th>Receive Size (bytes)</th>
<th>TCP Round-Trip (ms)</th>
<th>UDP Round-Trip (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>769</td>
<td>660</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>883</td>
<td>761</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>1024</td>
<td>868</td>
</tr>
</tbody>
</table>
UDP communications would enhance the speed of the µJini and VTC messaging system, but not by very much. UDP packets usually get to their destination and they did so for most of our tests. If UDP communication were to be implemented for the µJini system, measures would have to be taken to assure messages sent were received by the other peer and resent if necessary. A CurrentMessage field and RespondToMessage field have already been reserved in the µJini protocol classes for just this possibility.

Test Case 1

In the first test case, the time has been found for moving from the iDEN_Pot screen to the iCoffee_Map screen. Upon a keypress on the phone, the server should send the key event to the emulator, see the change of screens, and the VNC server should send the client the updated screen. The time found is from just after the keypress is sent to the server until after the updated screen is completely painted. Excessively out of range values have been eliminated from the test data, averages taken, and the results displayed in Table 11.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Update Size (bytes)</th>
<th>Server Time (ms)</th>
<th>Client Read (ms)</th>
<th>Client Paint (ms)</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hextile</td>
<td>979</td>
<td>0</td>
<td>1266</td>
<td>2508</td>
<td>3774</td>
</tr>
<tr>
<td>PNG 8-bit Color</td>
<td>821</td>
<td>71</td>
<td>2477</td>
<td>31</td>
<td>3838</td>
</tr>
<tr>
<td>PNG 8-bit Gray</td>
<td>563</td>
<td>47</td>
<td>554</td>
<td>33</td>
<td>1761</td>
</tr>
<tr>
<td>PNG 2-bit Gray</td>
<td>559</td>
<td>57</td>
<td>558</td>
<td>34</td>
<td>1799</td>
</tr>
</tbody>
</table>

The test data shows the results for four different encoding schemes, the number of bytes sent from the server to the client, the time the server spent processing the update, the time it took for the client to read in the update bytes, the time it took for the client to
paint the screen once an image was created, and the total time from keypress to a
cOMPLETE SCREEN REFRESH ON THE CLIENT SIDE. Note that there is no read time for the hextile
ENCODING. Hextile encoding is done by reading bytes as needed and drawing rectangles
as the data is read. The total time for hextile reading and painting is shown under the
CLIENT PAINT TIME HEADING.

Test Case 2

In the second test case, the time has been found for going from the MenuItem_1
SCREEN to the MenuItem_2 screen. Upon a keypress on the phone, the server should send
the key event to the emulator, see the change of screens, and the VNC server should send
the client the updated screen. The time found is from just after the keypress is sent to the
server until after the updated screen is completely painted. Excessively out of range
values have been eliminated from the test data, averages taken, and the results displayed
in Table 12. The test data is formulated in the table the same as in test case 1.

Table 12: Test Case 2 Performance Data

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Update Size (bytes)</th>
<th>Server Time (ms)</th>
<th>Client Read (ms)</th>
<th>Client Paint (ms)</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hextile</td>
<td>429</td>
<td>0</td>
<td>660</td>
<td>2778</td>
<td></td>
</tr>
<tr>
<td>PNG 8-bit Color</td>
<td>449</td>
<td>41</td>
<td>908</td>
<td>14</td>
<td>3011</td>
</tr>
<tr>
<td>PNG 8-bit Gray</td>
<td>355</td>
<td>36</td>
<td>422</td>
<td>14</td>
<td>2666</td>
</tr>
<tr>
<td>PNG 2-bit Gray</td>
<td>354</td>
<td>36</td>
<td>421</td>
<td>14</td>
<td>2568</td>
</tr>
</tbody>
</table>

Test Case 3

In the third test case, the time has been found for going from the Diagonal_1
SCREEN to the Diagonal_2 screen. Upon a keypress on the phone, the server should
change the screens and the VNC server should send the client the updated screen. The
time found is from just after the keypress is sent to the server until after the updated screen is completely painted. Excessively out of range values have been eliminated from the test data, averages taken, and the results displayed in Table 13. The test data is formulated in the table the same as in test case 1.

Table 13: Test Case 3 Performance Data

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Update Size (bytes)</th>
<th>Server Time (ms)</th>
<th>Client Read (ms)</th>
<th>Client Paint (ms)</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hextile</td>
<td>5538</td>
<td>2</td>
<td>--------</td>
<td>6586</td>
<td>8490</td>
</tr>
<tr>
<td>PNG 8-bit Color</td>
<td>572</td>
<td>61</td>
<td>1255</td>
<td>20</td>
<td>2458</td>
</tr>
<tr>
<td>PNG 8-bit Gray</td>
<td>470</td>
<td>31</td>
<td>464</td>
<td>20</td>
<td>1948</td>
</tr>
<tr>
<td>PNG 2-bit Gray</td>
<td>468</td>
<td>43</td>
<td>467</td>
<td>19</td>
<td>1980</td>
</tr>
</tbody>
</table>

Test Case 4

In the fourth test case, the time has been found for going from the Diagonal_2 screen to the Ovals screen. Upon a keypress on the phone, the server should change the screens and the VNC server should send the client the updated screen. The time found is from just after the keypress is sent to the server until after the updated screen is completely painted. Excessively out of range values have been eliminated from the test data, averages taken, and the results displayed in Table 14. The test data is formulated in the table the same as in section test case 1.

Table 14: Test Case 4 Performance Data

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Update Size (bytes)</th>
<th>Server Time (ms)</th>
<th>Client Read (ms)</th>
<th>Client Paint (ms)</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hextile</td>
<td>5403</td>
<td>2</td>
<td>--------</td>
<td>6499</td>
<td>8493</td>
</tr>
<tr>
<td>PNG 8-bit Color</td>
<td>1065</td>
<td>76</td>
<td>1353</td>
<td>19</td>
<td>2642</td>
</tr>
<tr>
<td>PNG 8-bit Gray</td>
<td>1004</td>
<td>46</td>
<td>524</td>
<td>18</td>
<td>1800</td>
</tr>
<tr>
<td>PNG 2-bit Gray</td>
<td>818</td>
<td>58</td>
<td>500</td>
<td>20</td>
<td>2104</td>
</tr>
</tbody>
</table>
Test Case 5

In the third test case, the time has been found for going from the Ovals screen to the Diagonal_1 screen. Upon a keypress on the phone, the server should change the screens and the VNC server should send the client the updated screen. The time found is from just after the keypress is sent to the server until after the updated screen is completely painted. Excessively out of range values have been eliminated from the test data, averages taken, and the results displayed in Table 15. The test data is formulated in the table the same as in section test case 1.

Table 15: Test Case 5 Performance Data

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Update Size (bytes)</th>
<th>Server Time (ms)</th>
<th>Client Read (ms)</th>
<th>Client Paint (ms)</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hextile</td>
<td>6071</td>
<td>2</td>
<td>--------</td>
<td>7261</td>
<td>8870</td>
</tr>
<tr>
<td>PNG 8-bit Color</td>
<td>534</td>
<td>59</td>
<td>1256</td>
<td>20</td>
<td>2552</td>
</tr>
<tr>
<td>PNG 8-bit Gray</td>
<td>486</td>
<td>31</td>
<td>471</td>
<td>19</td>
<td>1576</td>
</tr>
<tr>
<td>PNG 2-bit Gray</td>
<td>432</td>
<td>36</td>
<td>460</td>
<td>20</td>
<td>1883</td>
</tr>
</tbody>
</table>

Test Evaluation

From the network latency tests performed in the earlier section, it appears there is a substantial amount of delay found elsewhere in the screen update process. The server takes some time to process the update, but not a significant amount compared to the total time to complete the transaction. The phone is known to have delays in processing at times because it is a multi-function device and may have communication needs at the same time as the J2ME applications. The phone processor is also slower than a desktop machine and may be taking quite a while to create the PNG image from the stream. In addition, for color PNGs the image must be converted to a grayscale image to display on the phone that has a 2-bit grayscale LCD screen. Figure 12 shows the tests in a graph
scaled by the data size of the update. Figure 13 shows the tests in a graph depicted by the round-trip time of the update.

Figure 12: Test Data by Update Data Size

Figure 13: Test Data by Round-Trip Time
CHAPTER 6
ADDITIONAL WORK

This chapter describes some additional work performed to add more value to the µJini system. We describe a new streamlined protocol for appliances that defines some common objects needed in a user interface.

The VNC Appliance Protocol

Appliances (VCRs, TVs, microwaves, alarm systems, thermostats, soda machines, etc.) can easily implement a simple VNC server on a single low-cost semiconductor chip or just a few chips. This simple server can respond to key events from the client and send the client simple interface screen components. The VNC protocol is ideal for this since it is open source and client applications can be easily written. For a very simple device, this could be a state machine with no microprocessor. This allows a VNC server to consist of circuitry containing only a PAL, CPLD, or FPGA\(^1\) with a clock to drive the states. Commercially, this API could be used for soda machines, information kiosks, or many other devices. For an appliance interface there should be several simple components that are needed to present the interface: The appliance API we present here can be used to display panels, buttons, and menus to clients and receive input. The server

---

1 PAL (Programmable Array Logic), CPLD (Complex Programmable Logic Device), and FPGA (Field Programmable Gate array) are all chips that can be reprogrammed with simple logic similar to the concept of EEPROM devices. This is usually done with a language such as VHDL.
then sends back appropriate screen updates as needed for the client to see results. The attributes needed to represent an appliance object are show in the list below:

1. Screen size
2. Background color
3. Foreground color
4. Pen thickness
5. Line
6. Rectangle
7. Oval
8. Text
9. Button
10. List

The server can present these components to the client and it will be the client’s responsibility to display them in a reasonable format. By doing this there is more responsibility placed on the client that before, but allows a simpler VNC server. To further simplify the server, the server may choose to redraw the entire screen on an update instead of being restricted by the area requested by the client.

The initial handshaking for a VNC session will not change from the specifications. Only the *FrameBufferUpdate* message will be different and this is what is defined in this section. The first four bytes of a *FrameBufferUpdate* message contain the message type, padding, and the number of rectangle updates. In the new
FrameBufferUpdate format, the number of rectangles will be changed to number of objects where the object can be a rectangle or one of the new appliance UI objects we are defining in this section. This change is shown in Table 16.

Table 16: Beginning of FramebufferUpdate Message

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>message-type</td>
</tr>
<tr>
<td>1</td>
<td>Any</td>
<td>Padding</td>
</tr>
<tr>
<td>2</td>
<td>variable</td>
<td>number-of-objects</td>
</tr>
</tbody>
</table>

For each update object, there will be a data block starting with the header in Table 17. Encodings 7, 8, and 9 were defined for the µJini project and the encoding 11 is for the new appliance encoding defined in this section.

Table 17: Second Stage of FramebufferUpdate Message

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>variable</td>
<td>x-position</td>
</tr>
<tr>
<td>2</td>
<td>variable</td>
<td>y-position</td>
</tr>
<tr>
<td>2</td>
<td>variable</td>
<td>width</td>
</tr>
<tr>
<td>2</td>
<td>variable</td>
<td>height</td>
</tr>
<tr>
<td>4</td>
<td>variable</td>
<td>encoding-type</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Raw encoding</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Copy rectangle encoding</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>RRE encoding</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>CoRRE encoding</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Hextile encoding</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>PNG 8-bit Grayscale encoding</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>PNG 2-bit Grayscale encoding</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>PNG 8-bit color encoding</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Appliance encoding</td>
</tr>
</tbody>
</table>

For the new appliance encoding there will be an additional header that defines the type of object being transmitted as shown in Table 18.
Table 18: Appliance Object Description Value

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable</td>
<td>Type of appliance object</td>
</tr>
<tr>
<td>0</td>
<td>line</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>rectangle</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>oval</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>button</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>list</td>
<td></td>
</tr>
</tbody>
</table>

The different components will then be sent as shown in Table 19, Table 20, Table 21, Table 22, Table 23, Table 24, and Table 25.

Table 19: Appliance Line Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>foreground color</td>
</tr>
</tbody>
</table>

Table 20: Appliance Rectangle Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>foreground color</td>
</tr>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>background color</td>
</tr>
<tr>
<td>1</td>
<td>0(false) or 1(true)</td>
<td>fill area? (boolean)</td>
</tr>
<tr>
<td>1</td>
<td>variable</td>
<td>border thickness in pixels</td>
</tr>
</tbody>
</table>

Table 21: Appliance Oval Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>foreground color</td>
</tr>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>background color</td>
</tr>
<tr>
<td>1</td>
<td>0(false) or 1(true)</td>
<td>fill area? (boolean)</td>
</tr>
<tr>
<td>1</td>
<td>variable</td>
<td>border thickness in pixels</td>
</tr>
</tbody>
</table>
Table 22: Appliance Text Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>foreground color</td>
</tr>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>background color</td>
</tr>
<tr>
<td>variable</td>
<td>variable</td>
<td>Text String Object</td>
</tr>
</tbody>
</table>

Table 23: Appliance Button Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>foreground color</td>
</tr>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>background color</td>
</tr>
<tr>
<td>1</td>
<td>0(false) or 1(true)</td>
<td>Button has focus? (boolean)</td>
</tr>
<tr>
<td>variable</td>
<td>variable</td>
<td>Text String Object</td>
</tr>
</tbody>
</table>

Table 24: Appliance List Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>foreground color</td>
</tr>
<tr>
<td>(pixel depth) / 8</td>
<td>variable</td>
<td>background color</td>
</tr>
<tr>
<td>2</td>
<td>variable</td>
<td>number of items in list</td>
</tr>
<tr>
<td>variable</td>
<td>variable</td>
<td>Text String Object</td>
</tr>
</tbody>
</table>

Table 25: Text String Object

<table>
<thead>
<tr>
<th>NO. OF BYTES</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>variable</td>
<td>number of bytes in text following</td>
</tr>
<tr>
<td>n</td>
<td>variable</td>
<td>text string n bytes long</td>
</tr>
</tbody>
</table>

Appliance Example

One ideal situation for appliances will be inside a home network. From a mobile device or desktop computer outside the home, a user can gain access to many devices inside the home such as an alarm system, VCR, or thermostat controls. In this section is shown a thermostat appliance that presents and responds to a user interface using the VNC appliance protocol.
The New VNC Clients

The appliance API has been added to the VTC on the i85s device as well as being added to the java VNC viewer from AT&T [21] that executes using J2SE on desktop computers. The J2SE version can also be compiled for small handheld devices that support the necessary java libraries. To add the appliance API to the client code involves sending the new RFB protocol number for the appliance API during handshaking and interpreting the appliance API update objects as they are sent by the VNC server.

The Appliance API Device

The appliance API device can be built as a minimal circuit and possibly as a single chip if the device interface is simple enough. In Figure 14 is shown a circuit consisting of a clock, PAL, serial transceiver, and a block of NVRAM+SRAM. If enough space is left on the CPLD, the serial transceiver can be encoded there instead of using the additional chip. These components should be enough to build a very simple VNC server that supports the appliance API. There is no microprocessor in the circuit. The appliance is a simple state machine and performs actions dependent on the event and current state at the time of the event.

The Home Network and Appliance Server

The home network should have an ApplianceServer that authenticates the user and can validate input before sending it to the appliance. The ApplianceServer can handle many devices and can handle very simple devices by interpreting client messages and translating them into smaller proprietary messages if needed. The RS-232 protocol is shown only for this example. Any other transmission protocol can be used that is supported by both client and server such as Bluetooth or Wi-Fi wireless systems. Figure
15 shows an example layout of the home network. Clients will connect through the home firewall/router to the *ApplianceServer*.

![Figure 14: Possible Appliance API Server Circuit Components](image)

**Appliance API for Java Devices**

A simple set of methods has been implemented in the java language for use on a small device that has a JVM available. The java class containing the API calls is listed in Appendix C.

**The Thermostat Implementation**

A VNC Appliance API device has been created using the java APIs running on a Unix computer and the J2SE libraries. An emulator phone screen capture is shown in Figure 16 with the different API components labeled.
The title of the service is listed at the top of the screen as *Home Thermostat*. There is a line object below the title for aesthetic purposes. There are two buttons on the service: one for an on-off switch and another to switch between heating and cooling. Finally, there is a status line at the bottom that tells the current temperature surrounding the thermostat. This thermostat has two buttons with two options so there is four possible states the device must keep track of plus the current temperature that can be read from the memory of the device. The total size of a complete screen update in this example is 203 bytes.
Figure 16: Thermostat Display
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This thesis has shown a new way of dynamic service discovery and delivery for small devices with limited resources. By using a proxy and VNC server, any type of service can be found using many different service discovery protocols and returned to the client as a VNC server connection where the requested service is executing. By using this technique, a client must only understand and implement the µJini protocol and the VNC RFB protocols to be able to use services that a µJini proxy is capable of discovering. Another attribute of this architecture is the scalability; the allowance for newer, more powerful devices to use the system while still catering to the smaller, less powerful devices that will always exist and perhaps be more prevalent than the larger computing devices in the near future. The remaining value of the VTC system will be to the service provider by keeping the service code private and only executing on the server side. In addition, the service cannot be copied to another device for use by someone who has not purchased the service.

Conclusions

The µJini architecture succeeds in providing service discovery and delivery for small devices. The use of the VTC thin-client system provides a much faster startup time for the service over download, but the tradeoff is that user interaction with the service is noticeably hampered due to the latency of the 2G wireless network connection. As the communications industry moves into 3G and 4G wireless services, the network will be
much faster and the user interaction for VTC as well as the download and install times will be very acceptable. A VTC solution will always provide much faster access for users and will give service providers a way to keep control of their proprietary code.

Further Research

Additional research in this area may include support for a VNC proxy, more service discovery protocols, more service execution platforms, porting of the protocol and client to other languages, other network protocols, further enhancements to the RFB protocols, and further enhancements to the VNC server.

VNC Proxy

A VNC proxy can sit between the VNC server and client to perform translations and modifications to the RFB protocol. This may be of interest to someone wanting to include a new protocol such as the PNG protocol or differencing the images without opening up and modifying the source code of the VNC server. Not only is the code for the VNC server somewhat long and complex, but by modifying the source directly, the author is obligated to also share the source code of the new application because of the GPL license with which the VNC system is packaged. While open source can be great, there are many reasons why a business may wish to keep the source code private.

The main advantage to the proxy architecture is that it allows plug-in type filters for the VNC system. The VNC source code is known to work well and has been tested by many people in real applications. One VNC server can be started and many proxies can be run to implement many different protocols. The new VNC plug-ins can be written and sold commercially for profit while the source code remains private if the author chooses to do so.
An immediate disadvantage is that a proxy requires another point in the line of communication. Another piece in the communication line means additional time to process messages between client and server as well as another point of possible failure within the system.

Service Discovery Protocols

The proxy can be modified to support more service discovery protocols such as Salutation, UPnP, Jxta, and .NET. Additionally, private discovery protocols can be created on the server side to support in-house services and services that are hidden from public view.

Service Execution Platforms

This project used a Unix system to execute services as well as an emulator and JVM to execute MIDlets that required a J2ME environment. Additionally the proxy could support starting services on other platforms such as Windows, Apple, or Linux systems. The new .NET services that are expected to start appearing soon can be started on an additional Windows machine in addition to executing native Windows applications.

Porting of the Protocol and Client

The protocol, proxy, and client have all been implemented using the Java language. At the very least the protocol and client could be implemented in C/C++ for portability to many other small device platforms. They could also be ported to the new C# language for use with .NET applications and Windows machines.
UDP Communications

UDP communications could have some effect on the feel of an application as our network latency tests showed in the performance section of the thesis. Since UDP packets are usually reliable in today’s networks, we would expect to see a small speed increase in the user interface when using a VTC service with UDP communications. A downside to this will be the additional coding required on the client and server sides to implement reliability and message ordering with the UDP packets.

RFB Protocols and VNC server

More protocols can be created to make the image updates even more efficient for certain applications. The tests done in this thesis have shown that some update protocols are more efficient for certain images. We can develop new algorithms that make a best effort to find the most efficient algorithm and implement it as the update protocol for a particular screen update.
public boolean clientScenario() {

    MJBaseProtocol msg;
    MJProxyDiscoveryRequest mjprd;
    MJClientConnectRequest mjccr;
    MJServiceLookup mjsl;
    MJServiceLookupResponse mjslr;
    MJRequestVTC mjrvtc;
    MJRequestVTCResponse mjrvtcr;

    InputStream is;
    OutputStream os;

    // Create a request for proxies and multicast onto local network
    mjprd = new MJProxyDiscoveryRequest();
    sendUDP(mjprd, MJmsg.PROXY_MC_ADDRESS, MJmsg.PROXY_PORT);

    // Now get all proxies that responded to our multicast request
    String[] proxyList = getAllProxies();

    // If we found at least one proxy, let's use the first one to connect to
    if(proxyList.length > 0) {
        mjccr = new MJClientConnectRequest(curMsg++, nextMsg, RAMavailable,
                                            dataSpaceAvailable, programSpaceAvailable, 100, 111, 2,
                                            false, true, true, false);
        is = getInPutStream(proxyList[0]);
        os = getOutPutStream(proxyList[0]);
        mjccr.sendTo(os);
    } else {
        return false;
    }

    // Get the return message from the proxy
    msg = MJBaseProtocol.createFrom(is);
    if (msg instanceof MJAcceptClient) {
        // Proxy accepted our connection, so just continue
    } else if (msg instanceof MJDenyClient) {
        return false;
    } else {
        return false;
    }
mjsl = new MJServiceLookup(curMsg++, "MapService", null, null, 5,
RAMavailable, dataSpaceAvailable,
programSpaceAvailable));
mjsl.sendTo(os);

// Get the return message from the proxy
msg = MJBaseProtocol.createFrom(is);
if (msg instanceof MJServiceLookupResponse) {
    // Got the correct message, so continue
} else {
    return false;
}

// If we found at least one service, let’s use the first one found
if(mjslr.numFound > 0) {
    mjrvtc = new MJRequestVTC(mjslr.jarURL[0], mjslr.jadURL[0],
    mjslr.MIDletName[0], typeOfService[0]);
    mjrvtc.sendTo(os);
} else {
    return false;
}

// Get the return message from the proxy
msg = MJBaseProtocol.createFrom(is);
if (msg instanceof MJRequestVTCResponse) {
    // Got the correct message, so cast message and continue
    mjrvtcr = (MJRequestVTCResponse)msg;
} else {
    return false;
}

// Create new VTC with parameters we got back from proxy
vtc = new VtcViewer(this, myDisplay, mjrvtcr.VncIP,
    mjrvtcr.VncPort, rfbProto.EncodingPng2bit);
}
public boolean serverScenario() {

    MJBaseProtocol msg;
    MJProxyDiscoveryRequest mjpdreq;
    MJProxyDiscoveryResponse mjpdres;
    MJClientConnectRequest mjccr;
    MJServiceLookup mjsl;
    MJServiceLookupResponse mjslr;
    MJRequestVTC mjrvtc;
    MJRequestVTCResponse mjrvtcr;

    InputStream is;
    OutputStream os;

    Socket sock = acceptConnection();
    is = sock.getInputStream();
    os = sock.getOutputStream();

    // Get the discovery message from a client
    mjpdreq = (MJProxyDiscoveryRequest) MJBaseProtocol.createFrom(is);

    mjpdres = new MJProxyDiscoveryResponse();
    mjpres.sendTo(os);

    mjccr = (MJClientConnectRequest) MJBaseProtocol.createFrom(is);
    mjac = new MJAcceptClient();
    mjac.sendTo(os);

    mjsl = (MJServiceLookup)MJBaseProtocol.createFrom(is);

    // Do some discovery stuff
    performDiscoveryTasks(mjsl);
    // Send discovery responses from previous method call
    mjslr = new MJServiceLookupResponse(mjslr.jarURLs, Mjslr.jadURLs, 
                                          mjslr.MIDletNames, typeOfServices);
    mjslr.sendTo(os);

    // Get the return message from the proxy
    mjrvtc = (MJRequestVTC) MJBaseProtocol.createFrom(is);

    // Set up a VTC for client
    setUpVTC(mjrvtc);
// Send VTC request response to client
mjrvtcr = new MJRequestVTCResponse(VNCip, VNCport, EmulstorIP, EmulatorPort);
mjrvtcr.sendTo(os);
}
APPENDIX C
CLASS SKELETON FOR APPLIANCE API

// David Nordstedt
// VNC Appliance API helper utilities
// 2001.9.13
//
// ApplianceAPI.java

import java.io.*;
import java.util.*;
import java.text.*;

public class ApplianceAPI {

    public static final int LINE = 0;
    public static final int RECT = 1;
    public static final int OVAL = 2;
    public static final int TEXT = 3;
    public static final int BUTTON = 4;
    public static final int LIST = 5;

    // Write the update header to the given output stream
    public static void writeHeader(DataOutputStream dos, int x, int y,
                                    int width, int height, int encoding)
            throws IOException {
            dos.writeShort(x);
            dos.writeShort(y);
            dos.writeShort(width);
            dos.writeShort(height);
            dos.writeInt(encoding);
    }

    // Write the Line object to the given output stream
    public static void drawLine(DataOutputStream dos,
                                 int x, int y, int width, int height,
                                 int encoding,
                                 int color)
            throws IOException {
            writeHeader(dos,x,y,width,height,encoding);
            dos.writeByte(LINE);
            dos.writeByte(color);
    }
}
// Write the Rectangle object to the given output stream
public static void drawRect(DataOutputStream dos,
    int x, int y, int width, int height,
    int encoding,
    int foreGround, int backGround,
    boolean fillRect, int borderWidth)
    throws IOException {

    writeHeader(dos,x,y,width,height,encoding);
    dos.writeByte(RECT);
    dos.writeByte(foreGround);
    dos.writeByte(backGround);
    int fillVar = (fillRect == true) ? 1 : 0;
    dos.writeByte(fillVar);
    dos.writeByte(borderWidth);
}

// Write the Oval object to the given output stream
public static void Oval(DataOutputStream dos,
    int x, int y, int width, int height,
    int encoding,
    int foreGround, int backGround,
    boolean fillRect, int borderWidth)
    throws IOException {

    writeHeader(dos,x,y,width,height,encoding);
    dos.writeByte(OVAL);
    dos.writeByte(foreGround);
    dos.writeByte(backGround);
    int fillVar = (fillRect == true) ? 1 : 0;
    dos.writeByte(fillVar);
    dos.writeByte(borderWidth);
}

// Write the Text object to the given output stream
public static void drawText(DataOutputStream dos,
    int x, int y, int width, int height,
    int encoding,
    int foreGround, int backGround,
    String textString)
    throws IOException {

    writeHeader(dos,x,y,width,height,encoding);
    dos.writeByte(TEXT);
    dos.writeByte(foreGround);
    dos.writeByte(backGround);
    byte[] textBytes = textString.getBytes();
    dos.writeShort(textBytes.length);
    dos.write(textBytes, 0, textBytes.length);
}

// Write the Button object to the given output stream
public static void drawButton(DataOutputStream dos,
    int x, int y, int width, int height,
int encoding,
int foreGround, int backGround,
boolean hasFocus, String textString)
throws IOException {
    writeHeader(dos,x,y,width,height,encoding);
    dos.writeByte(BUTTON);
    dos.writeByte((byte)foreGround);
    dos.writeByte((byte)backGround);
    int focusVar = (hasFocus == true) ? 1 : 0;
    dos.writeByte(focusVar);
    byte[] textBytes = textString.getBytes();
    dos.writeShort(textBytes.length);
    dos.write(textBytes, 0, textBytes.length);
}

// Write the List object to the given output stream
public static void drawList(DataOutputStream dos,
    int x, int y, int width, int height,
    int encoding,
    int foreGround, int backGround,
    String[] textStrings)
throws IOException {
    writeHeader(dos,x,y,width,height,encoding);
    dos.writeByte(LIST);
    dos.writeByte((byte)foreGround);
    dos.writeByte((byte)backGround);
    for(int i = 0; i < textStrings.length; i++) {
        byte[] textBytes = textStrings[i].getBytes();
        dos.writeShort(textBytes.length);
        dos.write(textBytes, 0, textBytes.length);
    }
}

} // end class ApplianceAPI
REFERENCES


http://java.sun.com/aboutJava/communityprocess/final/jsr030/CLDCSpecification
1.0.zip, October 2001.


[16] Palm Boulevard, “Hardware: Palm m505,”
http://www.palmblvrd.com/hardware/Palm-Palm-m505-2001-3-19-palm-pc-


[18] NTT DoCoMo, “TECHNOLOGY -NTT DoCoMo R&D,”


Approach for Pervasive Computing,” MS thesis, University of Florida, August

BIOGRAPHICAL SKETCH

David Nordstedt was born on October 27th, 1965, in Columbus, Ohio, USA. He moved to Gainesville, FL, in 1969 when his father accepted a job at the University of Florida as a professor in the Agricultural Engineering Department. High school classes involved some college-level academic courses at Santa Fe Community College with some vocational training in automotive systems as well as heating and air conditioning. After finishing high school classes, he took a position as an apprentice cabinet-maker at Design Cabinets in Alachua, FL. After spending several years working through the different positions in the cabinet shop (sanding, cabinet-making, door production, finishing, hardware), he moved into an office position and was once again acquainted with computers. Always having been intrigued with electronics and computers during his earlier years, he resumed classes at Santa Fe Community College to finish his two-year degree in computer programming. Realizing that a much better job could be achieved with a four-year degree, he switched gears again and completed the AA degree before moving on to the University of Florida in 1998 and continuing to complete his B.S. in computer engineering in December 2000. He continued his studies at the University of Florida in the Computer and Information Sciences and Engineering Department and will receive his Master of Engineering degree in December 2001.

His personal interests away from school include music, art, film, theater, baking, electronics, and traveling. At some point he would still like to record a CD and perhaps open a jazz/blues/bakery/internet cafe.
His current research interests include wireless communications, mobile networks, and pervasive computing.