AN EXTENSIBLE COMMUNICATION-ORIENTED ROUTING ENVIRONMENT FOR PERVASIVE COMPUTING

by

Orlando Leon

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 24, 2002

Author_____

Department of Electrical Engineering and Computer Science

May 24, 2002

Certified by_____

Larry Rudolph

Principal Research Scientist

Thesis Supervisor

Accepted by_____

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

AN EXTENSIBLE COMMUNICATION-ORIENTED ROUTING ENVIRONMENT FOR PERVASIVE COMPUTING

by

Orlando Leon

Submitted to the

Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

The realm of pervasive computing is expanding at a fast pace, and the need for more generalized protocols and systems is in high demand. Bluetooth is a technology with great potential, but it runs over specific protocols standardized only for other Bluetooth devices. This limits connectivity and control of these devices. CORE, the communication-oriented routing environment, is a generalized message-based routing and inter-connect system that provides an implementation and framework for platform-independent, device-independent services and applications over TCP/IP. More importantly, CORE exposes a routing mechanism that allows extensions and abstractions through levels of indirection. Our implementation of CORE allows us to design an extension that will allow Bluetooth and non-Bluetooth devices to communicate, use, and control Bluetooth devices through CORE.

Thesis Supervisor: Larry Rudolph

Title: Principle Research Scientist

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Thanksgiving goes out to Dr. Larry Rudolph for all of his support, ideas, and visions throughout the year.

A big thank you goes to Brendan Kao, the co-designer and co-implementer of the CORE revision. Thanks also for co-writing parts of the related works section of this thesis and for some of the great graphics.

Thanks also to Shalini Agarwal, Amay Champaneria, Josh Jacobs, and Brendan Kao for encouragement, support, sympathy, and empathy as we struggled through this MEng year together.

I would like to thank my Rebecca Ng for the sympathy, support, encouragements, and consistent prayers throughout this year.

I thank my parents, Richard and Sandy Leon, for the support and encouragements the past twenty plus years.

I thank my church, BCEC, and my fellowship, Branch, for being my home away from home.

Most importantly, I thank God for bringing me to MIT and for faithfully providing during these years in Boston.

This work is supported in part by our Oxygen industrial partners, Acer, Delta, Hewlett Packard, NTT, Nokia, and Philips, as well as by DARPA through the Office of Naval Research contract number N66001-99-2-891702.

ACRONYMS

CLDC: Connected Limited Device Configuration

CORE: Communication-Oriented Routing Environment

DNS: Dynamic Naming System

INS: Intentional Naming System

MDD: Module Dependency Diagram

MIDP: Mobile Information Device Profile

RES: Remote Execution Server

RPC: Remote Procedure Call

TCP/IP: Transmission Control Protocol/Internet Protocol

UID: Universally Unique Identifier

UPnP: Universal Plug and Play

Chapter 1

Introduction

Imagine walking into the living room. We sit down on the couch and pull out our Bluetooth-enabled, rollable keyboard (see Figure 1), or possibly a rollable PDLCD touch-screen, from our pocket. Next, we start typing on the keyboard and the Tivo set starts playing our favorite sitcom show on the web-enabled TV. We then start typing on the keyboard again, but this time our favorite blues CD starts playing softly in the background. As we finish typing again, the automated, robotic dog brings us our evening newspaper that was just printed on the printer upstairs. No computers are in sight; the only computer in the house is in the basement.

Figure 1 - Rollable Keyboard

Today, there is no single technology that allows us to achieve such a scenario. One would have to plug in a keyboard directly into the device that is needed in order to start communicating. However, the traditional plug-and-play model does not fit well into a human-

centric world of pervasive computing. In this scenario, the one input device here can talk to all necessary devices via a wireless network. Correct packets of information are routed based on location and/or context of the input to the keyboard.

With CORE, a communication-oriented routing environment, we can combine several components to make this scenario possible. Provided we have Bluetooth-enabled keyboards and microphones, we can connect input devices that will be connected to and interpreted with syntactic interpretation (i.e. smart Rules, see section A.3.4) applications over CORE; these in turn will control devices such as TVs, VCRs, printers, and computers.

1.1 The Problem

Now, more than ever, and at a faster pace than ever before, computing power can be found in smaller and more common devices. We can find inexpensive, ubiquitous networking technologies everywhere. Moore's Law continues to take its effect on computing power, networking speed, and storage media. Soon, computing power will be as free and accessible as the oxygen all around us [7]. However, many devices and services speak their own languages and communicate with their own protocols. In a simple analogy, one can recall the story of the Tower of Babel told in the Holy Bible; many strong men could provide great resources, but no work could be done because they could not communicate with each other.

Bluetooth [19] is such a protocol-specific technology (see section 2.3) with a growing acceptance among manufacturers. It makes it easy to interconnect a whole range of Bluetooth-enabled devices replacing the current jumble of wires and cables. However it provides limited control over connectivity of protocol in peer-to-peer ad-hoc Bluetooth networks.

Current specifications and APIs will soon allow us to create an ad-hoc Bluetooth solution for the home. Take, for instance, a Bluetooth-enabled headset. Bluetooth-enabled devices in the home can be configured so that they perform a local area search for an audio-enabled output device by providing desired attributes. While walking through the house, upon discovering a compatible device (i.e. a headset) within range, a pairing will be made and outgoing audio will stream to the headset, such as from a television, a radio, and a phone. Now, suppose the phone rings, but the headset is not within range. The Bluetooth protocol requires that the headset be brought back into range before the phone is able to stream audio to the headset.

Wireless Ethernet with TCP/IP is a protocol that takes a different approach. Once a connection is set up, it is possible to consistently hear audio from a desired device to our wireless TCP/IP-enabled headset while walking through the entire house. We find TCP/IP devices by name (i.e. IP address). Even if out of range, TCP/IP has the ability to route packets to the correct location because of this naming. In this scenario, when the phone rings, the TCP/IP protocol requires the static address of the phone before being able to receive audio. With Bluetooth, we only need to be within range to be able to receive audio.

We want the best of both worlds, since the grass always seems greener on the other side. We do not want to carry around two headsets. Bluetooth benefits from the ability to locally discover devices on a general level with attributes. TCP/IP provides robustness in that when devices are out of range, packets still have the ability to be routed to the proper destination because of the protocol's naming scheme. Bluetooth lacks what TCP/IP offers.

With the increasing complexity and sheer number of different hardware devices and software services that exist today, humans are asked to perform more and more elaborate and

difficult tasks just to allow these devices and services to work and communicate with each other. Rather than providing some ad-hoc application-specific gateway technology, such labor intensive control jobs can easily and much more efficiently be performed by a platform and agent-independent control system. It is this objective that the CORE, the communication-oriented routing environment, system seeks to accomplish.

1.2 The Solution

CORE is a generalized, smart, decentralized routing system built on top of TCP/IP. It has the ability to take unspecific inputs along with a list of specified rules and route messages to their appropriate destinations. In addition, CORE also has the ability to pool resources with dynamic linking. More importantly, CORE exposes a routing mechanism that allows extensions and abstractions through levels of indirection. With this abstraction, Bluetooth devices can break its protocol limitation and be able to communicate with a wide range of devices and services through CORE[1]. Furthermore, CORE's generalized communication routing environment allows Bluetooth devices to potentially control and be controlled by any device, service, or application connected to CORE, including non-Bluetooth devices.

As we will see in the following chapters, the Bluetooth extension to CORE addresses the challenges with control issues and protocol-specific connections that plague Bluetooth devices.

1.3 The Roadmap

The rest of the paper is laid out as follows. Chapter 2 discusses motivations and related work in this area of research. This chapter also briefly discusses CORE, although a more

[1] Currently, there are Bluetooth gateways available, but these either allow for extended Bluetooth-to-Bluetooth communication or allow Bluetooth devices to access a local LAN.

thorough discussion can be found in Appendix A. In addition, this chapter presents background about the Bluetooth specification. Chapter 4 discusses the design and reference implementation of the Bluetooth extension to CORE. Chapter 5 gives a brief glimpse of possible future work with CORE. Finally, Chapter 6 concludes this paper.

It is important to note that the preliminary version of CORE was more of a reference implementation as it was limited by its design. For many reasons we concluded that it would be easier to redesign and reimplement CORE than to fix the preliminary version. Therefore, the design and implementation of the revision of CORE became a large part of the thesis work. We recommend a comprehensive understanding of this revision to CORE to help understand the Bluetooth extension to CORE; design and implementation details can be found in Appendix A and Appendix B. In addition, we assume a general understanding of networking and routing protocols and object-oriented programming in reading this paper.

Chapter 2

Background Work

The area of pervasive computing grows day-by-day because of the recognition of its importance, promise, and potential. Analyzing past and future potentials of computers and computing power, most recognize the need to move from a computer-centric paradigm to one that is human-centric.

The objective of this thesis is to present the design and implementation of a Bluetooth extension to CORE, which is a generalized routing scheme. The focus of CORE is to help shift from our computer-centric world to a human-centric one. The integration of Bluetooth and CORE gives the ability to use and control Bluetooth devices in the way we want instead of allowing standards and protocols to dictate what we can and cannot do.

2.1 Related Work

We admit that we are not the first ones on the pervasive computing and Bluetooth playing fields. However, with so many different aspects of possible research, there are bound to be overlaps with projects that have already been completed or currently in progress. The benefits are numerous since the pervasive computing community can learn from one another. The rest of this section gives brief summaries of some of these projects.

2.1.1 m-P@gent

m-P@gent is a framework developed to network and run environment-aware applications and to personalize content on resource limited mobile devices. Its infrastructure is comprised of three components: a core platform independent module, platform dependent add-on functional modules for different applications and types of devices, and a profile module that "derives requisite add-on modules from an environment keyword." The paper mentions working around resource limitations by having devices go through a home server for the exchange of modules when changing environments, thereby allowing for the addition and deletion of modules at certain times; however, these may not necessarily be dynamic at all times. It also mentions a great amount about the different versions of Java running on different devices and how to work around this possible issue to achieve compatibility and seamless networking. The examples of personalization of computing and physical environment seem good and useful, but the framework itself seems much more powerful.

The m-P@gent shows the possibility of expanding the field of pervasive computing to include devices that have limited resources. In addition, the framework provides more efficiency in regards to agent migration by providing better, more cooperative resource control consumption. The paper claims that this "framework will be a core software infrastructure for controlling networked appliances and building a pervasive computing environment." [1]

The mP@gent project provides a good framework and background of a system similar to CORE. mP@gent provides some resource management, and this is an aspect CORE will need to integrate in the future. Differences between mP@gent and CORE, however, lie in the routing mechanism and design of communication.

2.1.2 INS

The creators of the Intentional Naming System (INS) bill it as a "resource discovery and service location system for dynamic and mobile networks of devices and computers" [12]. The system comprises of an application overlay network of Intentional Name Resolvers (INR), which resolves requests for services. When a new service starts up and joins the network, the service gives a nearby INR a service description of itself in the form of a *name-specifier*. A client looking for a service sends another name-specifier to an INR, which replies with a list of services that satisfy the request.

There are three points to note that differentiate this project from a majority of other service discovery protocols. The first is that the method for describing and querying a service is done through an attribute-value tree, which is dubbed a name-specifier. Unlike other projects such as JINI [35], the INS name-specifier is very descriptive yet is constrained to a fixed structure.

The second differentiating characteristic is that the INRs form a non-hierarchical spanning-tree overlay network. Each INR contains exactly the same data (minus propagation delay), with updates the set of name-specifiers propagated throughout the tree. This design allows the INR to scale up in terms of the number of clients using the resolvers. This is important because of the third feature of INS.

The third and most unique idea is INS takes on the idea of integrating name/service lookup with the routing of the data packets to create a level of indirection between the nodes, which is called late binding. This means that a client can simply send data with the destination designated by a name-specifier rather than IP address. The advantage is that the client does not have to worry about how the device is connected. For example, the device can be a

mobile client that is moving from wireless access point to access point and changing IP addresses in the process. As the wireless device moves, it keeps an INR informed of its current address, so that any data packets that are destined to the mobile device are automatically routed to the new IP address. The INS system also supports two other types of routing: anycast and multicast, where anycast routes the data to any of the nodes that satisfy a given name-specifier, and multicast broadcasts the data to all of them [36]. For example, a printing service is only concerned with being able to print to a printer that supports certain capabilities, but beyond that it is neutral to the selection of the printer. So, the printing service can simply send out the file that needs to be printed and mark it for anycast to printers that support the necessary capabilities.

2.1.3 Java JINI and other JINI-related Projects

Sun Corporation's Remote Method Invocation (RMI), combined with Java's ability to download code and transmit class instances across the network, makes network utilization completely transparent to distributed applications. This gives the ability to actually change the way clients execute methods remotely, mostly transparent to the client itself.

RMI works as follows: A service that starts up locates a RMI registry, which acts as repository of known services so that clients can query it. The service gives to the RMI registry a service stub, which is an instance of the class that is responsible for marshalling calls to the service and unmarshalling responses. The file for the stub is sent over the network along with instance-specific data. All this is bound to a name in the RMI registry, by a name that the service specifies. When a client requests this service by name from the RMI registry, the stub along with instance data is returned to the client. The client then uses this stub to communicate directly with the service (see Figure 2) [28].

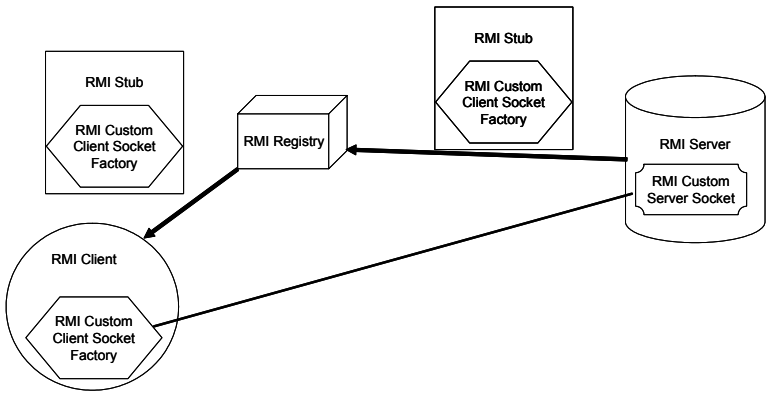

Figure 2 - RMI Client/Server Interaction

Sun created a layer on top of RMI, called JINI, to provide extra functionality and to make it possible to build agents that dynamically find each other and organize themselves into distributed applications. Two of these features are a more powerful and flexible lookup service and an explicit support for distributed events [29]. The discovery mechanism for JINI is much improved over RMI's. Instead of services being registered and looked up by a name as in RMI, which needs to be hard-coded, services can have associated attributes that are arbitrarily defined by the implementing service. Unlike, INS, there is no defined structure to these attributes; the only restriction is that the class that contains the attributes be a subclass of *Entry*. With JINI, the clients can now find desired services by querying for specific attributes without knowing the specific name of the service (please see the SDP paper for a detailed analysis of JINI's service discovery protocol [30]). The second important feature is the idea of distributed events, which allows the services to "push" data to clients. Whereas with RMI, the client is responsible for pulling (polling) data from the service, JINI now allows the servers to be the ones initiating the flow of data. This is not to say that distributed events cannot be implemented with RMI alone. In fact, it can, and this is how JINI accomplishes it. The important point is that the designers of JINI recognize the need for such a push model and have created explicit support for distributed events.

JavaSpaces is another project that builds services using the JINI infrastructure [31]. It is a data sharing mechanism that creates a virtual pool of data (known as *entry* in JavaSpaces nomenclature) where clients can insert data and retrieve data based on templates and types. This mechanism has the interesting property where clients that put data into this pool do not have any explicit knowledge of the consumer of that data, if any. A service responsible for processing this type of data can choose to process the data at any time, whether for load, arbitration, or any other reason. Likewise, the service that does take the entry form a JavaSpace needs to the origin of the generated entry. In fact, by the time a service processes the entry, the client that inserted the entry could already be disconnected form the JavaSpaces.

"[JavaSpaces] are designed to work with applications that can model themselves as flows of objects through one or more servers." [32] Essentially, the JavaSpaces acts as sort of a queue for the various services to take and insert entries. An analogy would be to treat JavaSpaces as a bulletin board; a client that requires a service posts its request and waits until a request can be processed.

2.1.4 Meta-Glue

In a lot of ways, the Metaglue [6] system is similar to the JINI system. Both have a lookup system, are based on Java and RMI, and have the aim of facilitating dynamically configurable systems. The difference, though, is that Metaglue aims to be a substrate for smart-spaces [33] as part of an over-arching program to build an intelligent room. To that end, it has several extra functionalities, such as the ability to move agents around from JVM to JVM, a network-accessible persistent backing store, and several others, but these are beyond the scope of this paper.

What is of particular interest is Rascal [34], which is a resource manager built into the Metaglue system. This resource manager provides both resource mapping and arbitration, meaning that Rascals know which resources can satisfy a particular client's request and can decide which client gets a particular resource when more than one client request the same type of resource. Rascal does all this via a knowledge base of each service's capabilities and needs, as well as each client's preference for resources when there is more than one satisfying resource. Using this knowledge, Rascal can compute the costs and benefits associated with particular arrangements of resources to clients. It then feeds all this information into JSolver, a constraint satisfaction engine, which then finds the "right" solution.

On top of this, a client which has been previously assigned a resource can be notified later by Rascal to stop using a particular resource, and perhaps to switch to another resource. One example of why this would happen is if a low priority background task requiring significant amounts of computing power -- perhaps as part of some maintenance -- is assigned a machine with lots of power. Along comes a user wishing to do a series of activities, such as watching movies, surfing web pages, etc. Rascal would determine that satisfying the users request is significantly more important than the background task, and hence the computing power would be reallocated for use by the user. As part of this process, the background task can be assigned to another computer which does not have as much computational power -- essentially a resource of smaller demand. Of course, the penalty incurred by changing a previously allocated resource is part of the previously mentioned calculations of cost and benefit.

In short, with Metaglue a client does not control the other end of its communication, and instead delegates that responsibility to a third party. This feature is unique from previously mentioned projects because it incorporates the concept that the client can be instructed about its communication partner at any time. It is also important to note that this is a "manager" of

resources, not merely a mechanism for "discovery", meaning that a client needs to carry through with what the manager says and not decide to use another resource that the manager did not approve for it to use. The method that this resource management is implemented with actually involves client stubs, making this entire process pretty transparent to both the resource provider and consumer.

2.1.5 Anhinga Project

The Rochester Institute of Technology (RIT) built a subset of JINI and coined it the JINI Micro Edition. In addition they wrote their own implementation of JavaSpaces. In short, Anhinga is a distributed, decentralized infrastructure created to support many-to-many applications designed to run on ad-hoc networks of small, mobile, wireless devices. The infrastructure creates a broadcast environment based on messages and is meant for lightweight versions of Java, JINI, and JavaSpaces. One implementation detail worth mentioning is that the Java 2 Micro Edition does not have mobile code moving, so the developers at RIT added it to their version of code. [8] [11]

RIT designed this project with small, wireless devices in mind. They have a Bluetooth extension as well, and it shows how to bridge Bluetooth devices with JINI.

2.1.6 Universal Plug-and-Play

Universal Plug and Play (UPnP) is an extension to the Device Plug and Play (PnP) technology that exists on personal computers today. UPnP excels over PnP in that it allows for a more generalized way to set up, configure, and add peripherals to a networked device that can use the TCP/IP and other standard internet protocols. Some devices include networked printers, handheld PDAs, and laptop computers.

Some other great ideas in this specification are that it is designed to support "zero-configuration, "invisible" networking and automatic discovery for a breadth of device categories from a wide range of vendors." In short, this means that if the whole world followed this specification for all future devices, devices should be able to connect almost seamlessly and with little, if any, user interaction.

Devices following the UPnP specification will be able to dynamically join a local network, obtain an IP address, broadcast its specs and capabilities, and perform searches of the local area to learn the presence of other devices. This is all done automatically upon entering a network thus truly enabling zero configuration networks. Connected devices can then communicate directly, enabling peer-to-peer and ad-hoc networking. Furthermore, since UPnP uses standard TCP/IP and Internet protocols, UPnP devices should be able to seamlessly integrate into existing networks.

The greatest benefit that comes from UPnP is that it is platform-independent and language-independent. In addition, UPnP is a distributed, open-networked architecture defined by the protocols used UPnP and does not specify application APIs. This allows for more flexibility and extensibility, though it also means that applications from different companies may or may not be able to communicate and integrate as easily. UPnP targets a wide array of users from the home, to the office, and even to corporations.

Microsoft makes the following claims about UPnP as "universal" [13]:

- **Media and device independence.** UPnP technology can run on any medium including phone line, power line, Ethernet, RF, and 1394.

- **Platform independence.** Vendors use any operating system and any programming language to build UPnP products.
- **Internet-based technologies.** UPnP technology is built upon IP, TCP, UDP, HTTP, and XML, among others.
- **UI Control.** UPnP architecture enables vendor control over device user interface and interaction using the browser.
- **Programmatic control.** UPnP architecture also enables conventional application programmatic control.
- **Common base protocols.** Vendors agree on base protocol sets on a per-device basis.
- **Extendable.** Each UPnP product can have value-added services layered on top of the basic device architecture by the individual manufacturers.

2.2 CORE

CORE, a communication-oriented routing environment, is a smart, generalized device-interconnect and routing system built on top of TCP/IP. It allows for platform-independent devices, services, and applications to send messages and potentially use other services connected to CORE. It has the ability to take unspecific inputs along with a list of specified rules and route messages to their appropriate destinations.

CORE takes an attempt at combining devices and pooling resources to create an environment where different devices with different protocols can communicate with the minimum of requirements. Figure 3 depicts applications and services running possibly different operating systems and form factors connecting to one CORE system. They are able to communicate with each other and share resources despite the differences between them.

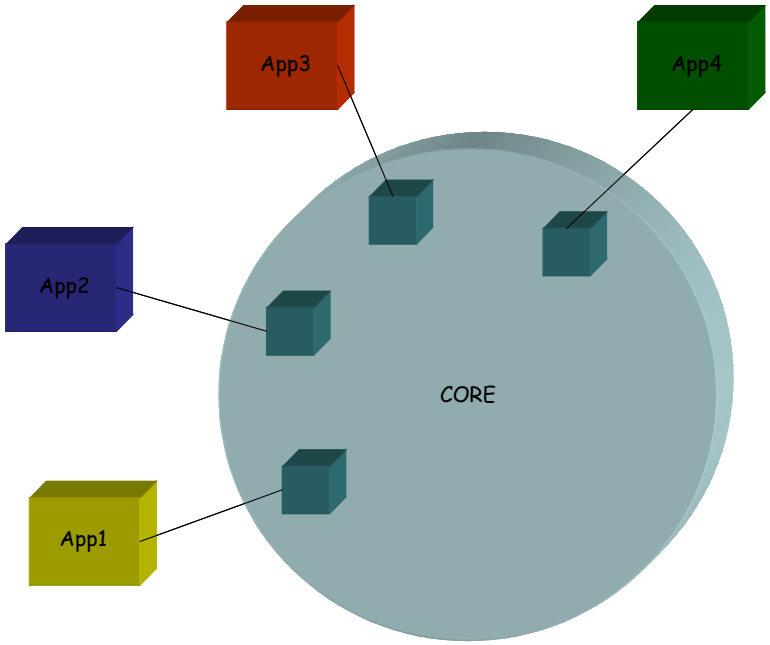

Figure 3 - CORE and Applications/Services

The goal of CORE is to create a system that is human-centric in its use and control. In other words, the CORE system allows the user to perform simple as well as complicated computational tasks using a variety of different non-homologous hardware and software agents together without having to understand and rewrite code and other issues that arise from linking devices and services that were not created to work together automatically (Figure 4 demonstrates a possible setup of a home entertainment center running through CORE; this can be compared with the generalized view of CORE in Figure 3). We designed and implemented CORE with extensibility as one of the main focuses, and the Bluetooth extension discussed in this paper is just one of the many possible extensions to CORE.

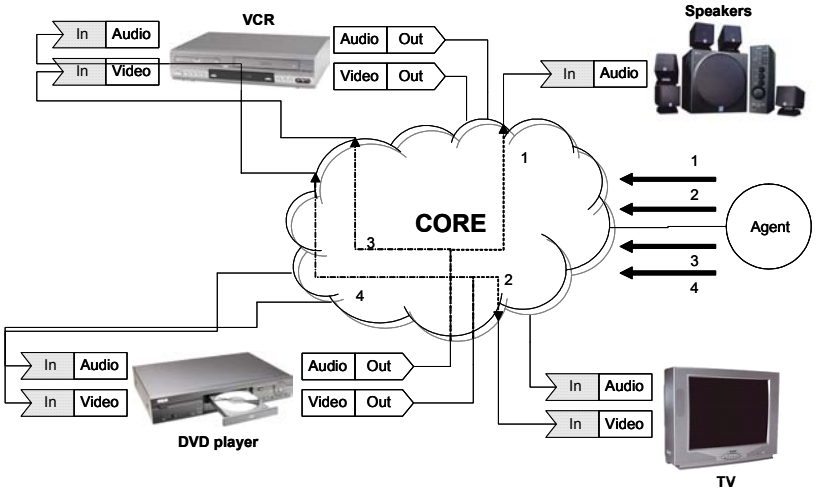

Figure 4 - Possible Scenario of a Home Entertainment Center
Running Through CORE

2.3 Bluetooth

Bluetooth is a wireless technology with many skeptics, but many are hopeful of its promising potential. Many analysts see Bluetooth devices being more pervasive than 802.11b devices in the near future. However, concerns with bandwidth limitation and interoperability with other devices have kept many analysts skeptical.

Integrating Bluetooth with CORE may show that some of these concerns can be overcome. With this extension to CORE, there are benefits for both Bluetooth and CORE. For CORE, nodes can now use Bluetooth devices, which are not able to directly communicate via TCP/IP. For Bluetooth, devices can now communicate and use services on CORE, which can be any device able to communicate over TCP/IP.

2.3.1 What is Bluetooth

It is necessary to cover some background detail of the Bluetooth specification before discussing details of the CORE-Bluetooth extension design and implementation. It is also

helpful to understand our design and implementation of CORE, which can be found in Appendix A and Appendix B.

The creators of the Bluetooth protocol wrote the specifications of Bluetooth with the intention of making it a solution for replacing cables. Take for instance a home entertainment center. Setting up an entertainment center can take hours because each component has its own set of wires and cables, some with both positive and negative connections (for example, see Figure 5). This can include a CD player, TV, VCR, DVD player, speakers, subwoofer, receiver, cable box, satellite box, DirectTV box, VCD player, and possible much more. This can easily exceed a hundred wires depending on the complexity of the setup. Of course, Bluetooth, as it is, is not meant to replace *all* wires and connections. Some connections require extremely fast data transfer rates, such as DVD players, and the current version of Bluetooth is limited to approximately 1Mbps. This is much too slow to provide high-quality, high-volume streams of information; however, the current speed is sufficient for voice/sound communication and possibly video-conferencing. Bluetooth starts us on this path of universalizing the method in which our electronic parts to communicate with one another.

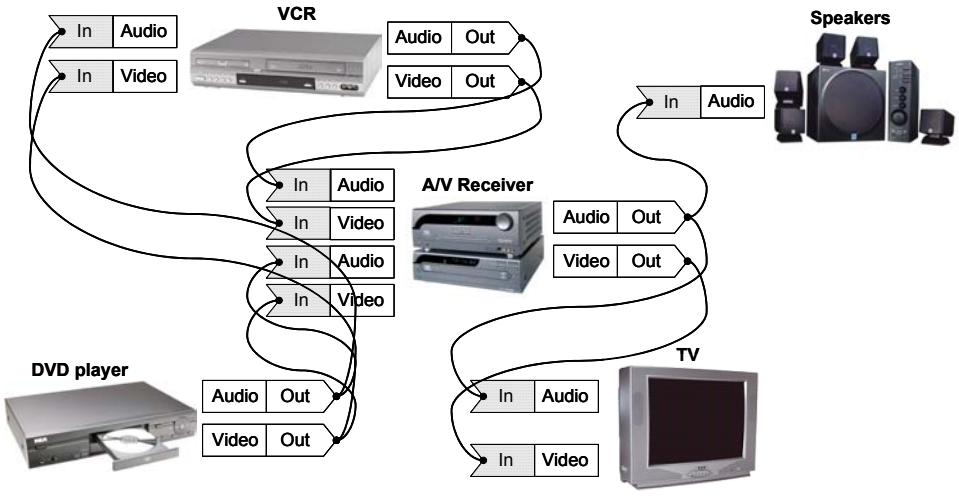

Figure 5 - Basic Home Entertainment Setup with Basic Connections

In the realm of CORE, the higher-level view of this abstraction is that it provides a gateway to communicate with Bluetooth devices. On the lower level, CORE provides a level of indirection and abstracts the communication details between Bluetooth-to-Bluetooth and Bluetooth-to-TCP/IP devices. This extension allows for a whole world of possibilities in device interoperability and communication. For example, whereas CORE previously could not accept input directly from a Bluetooth-enabled keyboard, this extension makes it possible. Bluetooth combined with CORE provides a more powerful means of communication for devices to communicate, both with and without cables and from possibly anywhere in the world.

2.3.2 Basics

Bluetooth is a standard that allows any sort of electronic equipment, such as computers, cell phones, keyboards, and headsets, to make connections without any wires and cables and without direct user-interaction, if desired. Bluetooth is designed to work on the two following levels:

- Physical level: Bluetooth is a radio-frequency standard
- Protocol level: the Bluetooth standard defines how data is sent, how much data is sent, and how parties can ensure correct data is sent

Another standard of wireless communication is the Infrared Data Association (IrDA). IrDA communicates with beams of infrared light and is differentiated with different frequencies of light pulses. IrDA is fairly reliable; however, there are two immediate drawbacks. First, IrDA is a line-of-sight method of communication. This means that the transceiver and receiver must be able to "see" each other within a certain angle. Next, IrDA is mostly a one-to-one communication in that only two devices can talk to each other at any

given time using IrDA. These qualities can be advantageous in that there is less chance of interference and that it ensures more privacy with the one-to-one limitation [19].

HomeRF and IEEE 802.11 are two other wireless standards aimed at advancing the wireless networking initiative. Both of these specifications can achieve higher data rates and have more networking features than Bluetooth. However, Bluetooth does not aim to support a complete LAN as Ethernet does; instead, Bluetooth is targeted to replace wires in smaller devices. In addition, Bluetooth is also aiming for low power requirements and low cost. These three technologies target the same general arena and paradigm, but Bluetooth targets a more specific level of wireless devices, mainly smaller devices with limited power.

With Bluetooth, there are three main features worth summarizing:

- *It is wireless*: when leaving the home or office, wires and connections are unnecessary
- *It is inexpensive*: with Bluetooth chip costs down to about $5, it is possible to have Bluetooth integrated into most electronic devices
- *It can be automated*: users are not required to interact with Bluetooth devices to get them talking with one another

2.3.3 Interference

On the radio frequency spectrum, Bluetooth communicates within 2.45GHz frequency range. There are currently a number of other devices types that also use this frequency, and this can possibly cause interference. Baby monitors, 2.4GHz cordless phones, microwaves, garage door openers, and 802.11b communicate on the same frequency. This presents a possible interference problem, especially in a wireless LAN or a typical modernized home.

One help in this matter is that Bluetooth operates at a low power output level; this limits the transmission range of devices. Bluetooth devices also perform what is called "spread spectrum frequency hopping." In this technique, Bluetooth devices choose 79 random frequencies to hop between at 1600 times per second. This method lessens the probability of interference between non-Bluetooth and Bluetooth devices alike, because any possible interference lasts only a tiny fraction of a second.

A recent paper analyzed interference between IEEE 802.11 and Bluetooth devices in a wireless medical environment [21]. The paper concludes that there is indeed interference between the two technologies, but if strategically placed interference can be minimized.

2.3.4 Specifications of Operation

Bluetooth typically operates at the low power of 1 mW. At this low power setting, it has a typical range of 10 meters, or about 30 feet. Two other power modes are offered: 2.5mW and 100mW, which gives 20 meters and 100 meters, respectively. Since it operates on the 2.45GHz radio frequency range, walls and other objects will not block transmission as it would for IrDA, for example.

As mentioned above, Bluetooth devices choose and hop on 79 independently chosen frequencies within the allowed spectrum. When Bluetooth devices come in range with one another, they automatically or manually communicate with each other to determine if a connection should be established. Two or more devices choosing to form a connection is called a personal area network (PAN), or a piconet. Devices in the same piconet then randomly hop frequencies in unison to stay in touch with each other and to avoid interference with other piconets.

When two or more independent and non-synchronized piconets communicate with each other, it is called a scatternet [14] (see Figure 6). Whether a device is a slave or a master in one piconet, it can become a slave in another piconet. These two piconets can then relay information between the two piconets if desired. However, a device cannot be a master in two different piconets. If a master in one piconet wants to create another piconet and become the master of that piconet as well, the Bluetooth specification provides primitives that make it possible to dynamically change master in a piconet during operation; this allows master-ship to be delegated to another device in the piconet.

Bluetooth devices have the ability to communicate with one another in half-duplex or full-duplex. For instance, a typical telephone is a full-duplex device, while a typical walkie-talkie is a half-duplex device. Bluetooth devices can send 64,000 bits per second in full-duplex mode; this rate is good enough to support several human voice conversations. Two types of half-duplex modes are available. The first mode is not proportional; data can flow at 721 Kbps in one direction and 57.6 Kbps in the other direction. The second mode is symmetrical; data can flow at 432.6 Kbps in both directions.

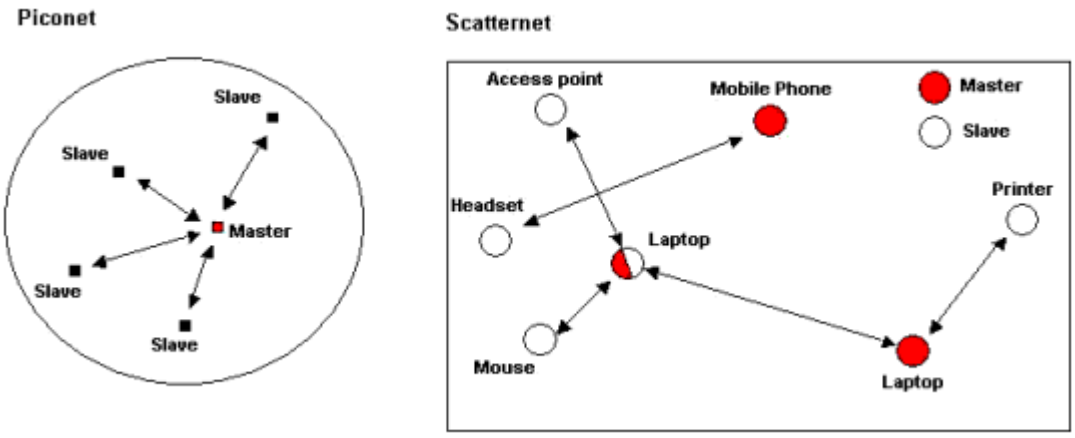

Figure 6 - Bluetooth Piconet (left) and Scatternet (right)

Devices in the same piconet share a common data channel. The total capacity is 1Mbps, and headers and overhead consume about 20% of this bandwidth. As mentioned above, devices randomly hop in unison at 1600 times per second on 79 randomly chosen 1MHz bandwidth frequencies. This translates to 625μs per timeslot. A piconet can have one master and up to seven slaves; masters transmit on even time slots while the slaves transmit on odd time slots. Packets are allowed to be up to five time slots wide; one data packet can be up to 2,745 bits in length.

Currently there are two types of data transfer allowed: synchronous connection oriented (SCO) and asynchronous connectionless (ACL). ACL provides a packet-switched connection between the master and all active slaves; a polling scheme is used to traffic from slaves to the master. An ACL link is either point-to-point or a multi-point broadcast. A master and a slave can have one ACL link. In addition, ACL slaves can only transmit when requested by the master. SCO is a point-to-point connection between one master and one slave. In a piconet there can be up to up to three SCO links running at 64,000 bits per second.

The Bluetooth specification contains three core connection protocols: the logical link control and adaptation protocol (L2CAP), the service discovery protocol (SDP) and the RFCOMM protocol. L2CAP provides data services to the higher layer protocols with protocol multiplexing capability, segmentation, and reassembly operations and group abstractions. Device information, services and the characteristics of the services can be queried using the SDP.

Like SDP, RFCOMM is layered on top of the L2CAP. As a cable replacement protocol, RFCOMM provides transport capabilities for high-level services that use serial line as the transport mechanism.

2.3.5 Establishing a Network Connection

We have described the different kinds of networks and types of links in a Bluetooth PAN. This section discusses the steps in service discovery and connection.

All devices start in a "standby" mode (see Figure 7) prior to being connected. While in this mode, devices "wake up" every 1.28 or 2.56 seconds (depending on selected option) to "listen" for messages on a set of 32 of its chosen hop frequencies.

Devices desiring to establish connections take initiative by sending a "page" message, if the address is already known; otherwise, an "inquiry" message is sent followed by a "page" message (similar to CORE's QUERY message followed by a RULE message desiring to establish a link; see sections A.3.3.1 and B.1.2). This device subsequently becomes the master of the connection.

While sending a "page," the master sends 16 identical, consecutive messages on 16 different frequencies for the slave device. If there is no response, then the master transmits on the remaining 16 frequencies. The maximum time before the master communicates with the slave is twice the length of the wakeup period for the slave, which is 2.56 or 5.12 seconds (depending on selected option). The average time to communication is only half the wakeup period, which is 0.64 seconds.

A device sends an "inquiry" message when it wishes to find Bluetooth devices with certain parameters and attributes. The sequence of outgoing messages from the master device is similar to the "page" request, except that there can be an additional period to collect responses.

While devices are connected, they can enter a power-saving mode to reduce power usage. Devices can enter this if no data needs to be transmitted in a piconet. The master device can

request for the slave to enter a "hold" mode; the slave device only has an internal timer running. On the other hand, slave devices can demand to be put in "hold" mode. Once out of the "hold" mode, data transfer resumes as normal. In general, this mode is used when connecting piconets together or when managing low-power devices.

Two other low-power modes are also available: "sniff" and "park" modes. "Sniff" mode is characterized by a slave device listening on the piconet at a reduced rate; this rate is programmable. "Park" mode is characterized by slave devices being passive members in the piconet; these devices do not participate in the traffic. Furthermore, these devices relinquish their Bluetooth MAC addresses; however, they still occasionally listen to traffic from the master to resynchronize and listen for "page" messages.

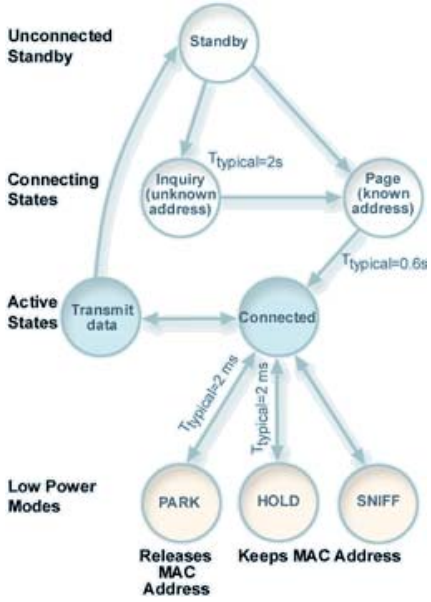

Figure 7 - Connection State of Bluetooth Device

Figure 7 shows a state machine of Bluetooth devices in service, discovery, and connection.

2.3.6 Packet Definition

A packet consists of three fields: a 72-bit access code, a 54-bit header, and a payload of variable length (2-342 bytes) (see Figure 8).

72	54	16-2745
ACCESS CODE	HEADER	PAYLOAD

Figure 8 - Bluetooth Packet

The access code is used for synchronization, DC offset compensation, and for identification. All packets sent within the same piconet *must* have the same access code. Figure 9 shows the access code segment of the Bluetooth packet in more detail. The sync word is derived from the lower address part of the master's 48-bit address.

4	64	4
PREAMBLE	SYNC WORD	TRAILER

Figure 9 - Access Code of Bluetooth Packet

The header of the Bluetooth packet contains lower-level link information. It consists of the following six fields: a 3-bit sub address[2] (M_ADDR), a 4-bit packet type (TYPE), a 1-bit flow control bit (FLOW), a 1-bit acknowledge indication (ARQN), a 1-bit sequence number (SEQN), and an 8-bit header error check (HEC). This gives us 18 bits; the remaining 36 bits are used for forward-error correction coding.

[2] The 3-bit address is the limitation that only allows for 8 devices; this is where we get the one master and seven slave devices in a piconet

3	4	1	1	1	8
M_ADDR	TYPE	FLOW	ARQN	SEQN	HEC

Figure 10 - Header of a Bluetooth Packet

The 4 bits of the packet field gives to 16 different packet types. 4 packets are reserved for control packets common to all physical link types. 6 packets are reserved for packets occupying a single time slot. 4 packets are reserved for packets occupying three time slots. Finally, 2 packets are reserved for packets occupying five time slots (see Figure 11).

Segment	**TYPE**	**SCO link**	**ACL link**
Control Pakets	0000	NULL	NULL
	0001	POLL	POLL
	0010	FHS	FHS
	0011	DM1	DM1
Single Slot Packets	0100		DH1
	0101	HV1	
	0110	HV2	
	0111	HV3	
	1000	DV	
	1001		AUX1
3-Slot Packets	1010		DM3
	1011		DH3
	1100		
	1101		
5-Slot Packets	1110		DM5
	1111		DH5

Figure 11 - Bluetooth Packet Types

Chapter 3

Bluetooth-CORE Integration

This chapter outlines our extension of CORE to integrate Bluetooth. The extension allows nodes using CORE to experience benefits of Bluetooth device services, such as Bluetooth-enabled keyboards, headsets, and cell phones. However, the incompatibilities between Bluetooth and TCP/IP make integration a challenge.

Since not all Bluetooth devices are Java-enabled and lack the ability to directly communicate over TCP/IP, direct communication with CORE is impossible. Even the Bluetooth devices that have Java capabilities, they may still be limited to certain profiles of the Java 2 Micro Edition, such as CLDC or MIDP; this limits networking capabilities and makes communication with CORE challenging.

The rest of this chapter discusses the design to extend CORE and gives an analysis of the challenges and benefits presented thereof.

3.1 Overview of Design

We extend CORE by creating a module to allow Bluetooth devices to access CORE and visa versa. It is required that the extension is easy to use, extensible, and scalable. In addition, this system should integrate easily with little or no modification to CORE.

Consider the Bluetooth protocol stack in Figure 12. It is possible to emulate TCP/IP over RFCOMM; however, Bluetooth devices cannot use this capability directly without some sort of bridge or gateway. There currently are devices available that provide this sort of bridging

[25], but they only bridge by providing LAN (i.e. TCP/IP) access, for example, whereas CORE can provide additional services in addition. Furthermore, commercial gateways currently cost around $3,000.

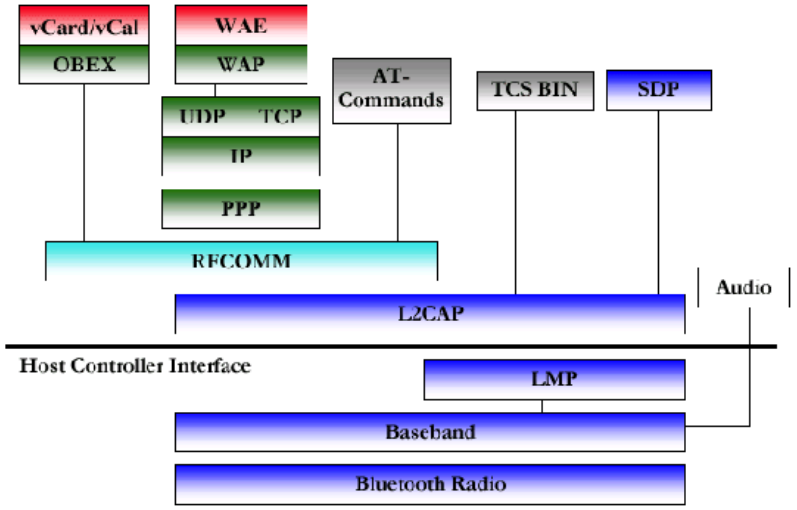

Figure 12 - Bluetooth Protocol Stack

To overcome this challenge and integrate Bluetooth into CORE, we add a class, BluetoothAgent, which is similar to the Agent class (see Appendix section B.1). The main function of this class is to serve as a Bluetooth listener for requests to connect to CORE. This class spawns off a thread for each incoming request; each thread is analogous to the Agent class serving an application. In addition, this class also serves as a Bluetooth discovery service for CORE. As a future extension, we can modify CORE to expose this ability to discover Bluetooth devices through the BluetoothAgent class.

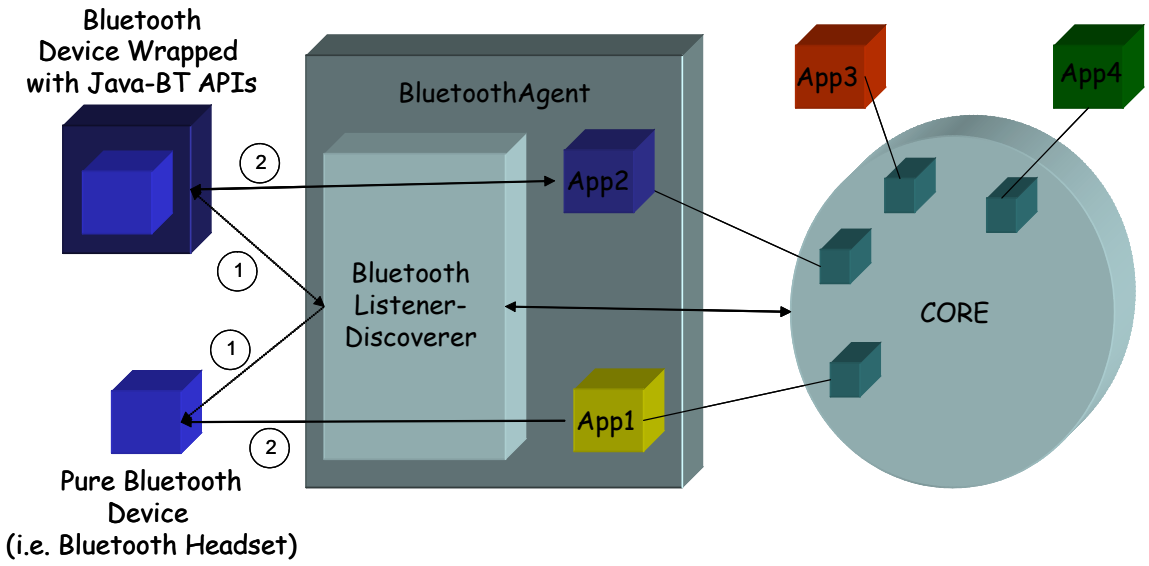

Figure 13 - Pictoral Representation of BluetoothAgent and CORE

Figure 13 shows CORE with an attached BluetoothAgent module. This can be compared to Figure 15 in Appendix A. Here, applications 3 and 4 can be any Java-enabled application or service that can communicate directly to CORE via TCP/IP[3]. The upper Bluetooth application/device[4] connects to CORE via a stub that BluetoothAgent provides. The Bluetooth device first connects to the BluetoothAgent in search of a CORE server; once connected, the Bluetooth device talks as if it is talking directly to CORE (i.e. black box). The lower Bluetooth device is not wrapped with a Java API (such as a Bluetooth headset), so it cannot talk to the BluetoothAgent or CORE directly. CORE sends BluetoothAgent a query to search for a certain device by giving attributes, such as a NameSpecifier (see section A.3.1). If BluetoothAgent finds a valid device, it spawns off a thread to handle the new client device. From here on, CORE thinks it communicates directly with the Bluetooth device (i.e. black box).

[3] One future extension will be to standardize serialization and deserialization of messages to allow non-Java devices to also communicate over TCP/IP (see section 4.3).

3.2 Implementation

This section discusses some implementation details as well as challenges faced and the future for our research group in this field. We made use of several Bluetooth devices from a few different manufacturers: an Anycom CF Bluetooth card, a 3Com PCMCIA Bluetooth card, a Compaq iPAQ 3870 with built-in Bluetooth, and an Ericsson Bluetooth headset. We also attempted to communicate with these devices using a Java Bluetooth stack and an implementation of JSR-82. Of the companies we explored, we could not find a free, working implementation of a Bluetooth stack (see section 4.10), but we chose the JSR-82 implementation from Atinav Inc [20].

Our reference implementation for the BluetoothAgent is written but untested since we are unable to obtain a free, functioning version of a Java-Bluetooth stack. The reference implementation contains code with in-line comments and questions for future reference. The code compiles since we have reference implementations of JSR-82; however, these JSR-82 reference implementations only contain class files and not the source files, so we cannot learn from the source code.

Our reference implementation of the Bluetooth extension to CORE contains two classes: BluetoothAgent and BluetoothClient. BluetoothAgent is the module that connects to CORE as described in the previous section. BluetoothClient is the wrapper for Bluetooth devices to use to connect to CORE via the BluetoothAgent.

---

[4] From here on we will refer to a Bluetooth application, device, or service as a device.

BluetoothAgent serves as a Bluetooth gateway to CORE. This class acts as another service application of CORE. It connects to CORE via the Agent class and identifies itself as a BluetoothAgent service. Aside from that, it can also serve CORE as a Bluetooth service discovery agent. With a slight extension to CORE, it can send queries to the BluetoothAgent in hopes to find a Bluetooth device that it seeks. If the query is successful, BluetoothAgent spawns off an Agent (see section Appendix B) class and connects the Bluetooth device to CORE.

After connecting to CORE, BluetoothAgent sits as a Bluetooth server waiting to be discovered by other Bluetooth devices searching for a CORE server. Once a connection is established, BluetoothAgent spawns off an Agent class to allow the Bluetooth device to "directly" communicate to CORE. As mentioned before, this CORE extension abstracts these connections and exposes Bluetooth devices to CORE through a level of indirection.

The BluetoothClient class serves as a wrapper for any Bluetooth device that can use Java, whether J2SE or J2ME and one of its profiles. Upon execution, the Bluetooth device loops and performs a service discovery search in its local area searching for a "CORE" server. Once found, it connects and sends its device/service characteristic to be used later by BluetoothAgent in a NameSpecifier. From here on, the Bluetooth device can implement whatever it needs to send and receive messages to and from CORE (see section B.2 and source code for examples).

Sections C.3 and C.4 in the appendix contain the source code for the reference implementations of BluetoothAgent and BluetoothClient.

3.3 Analysis

Bluetooth, IEEE 802.11, and HomeRF all have the capability of wireless networking. However, the main focus of Bluetooth is more of a cable replacement technology than anything else. These three technologies can be seen as complementary rather than competing. However, Bluetooth has the power and cost advantages, so it is very possible that Bluetooth will become more pervasive more quickly than the other two technologies.

The integration of Bluetooth devices into CORE was based on the potential benefits of the Bluetooth technology. Bluetooth devices offer characteristics, such as being wireless, fast, unconstrained by line-of-sight, cheap, able to connect a broad range of devices together, highly standardized, and widely-compatible. However, even though the Bluetooth specifications are highly standardized, its developing environment is not quite at the same level of standardization. As more and more Bluetooth stacks and JSR-82 implementations roll out, we will see a convergence. It is not until then that we will see more use for Bluetooth in the pervasive computing world.

In terms of Bluetooth specifics, the extension to CORE is not able to extend the 7 device limit in a piconet. However, as people conduct more research on the topic of scatternets, this limit will no longer be a concern since we can make a slave device into a master device in another piconet to extend the 7 devices to 14 devices. Using this approach, this type of ad-hoc networking makes the number of connections almost limitless.

In terms of possible challenges, it is notable to mention a few. First of all, packet and payload limitations may limit the usefulness of this extension to CORE. Theoretically, Bluetooth devices can support streaming audio and video, but we cannot expect the quality to be the same as streaming media over Ethernet.

Interference may give rise to another concern (see section 2.3.3). As more devices move toward a wireless environment, there will possibly be more interference with Bluetooth devices. On possible workaround is to select a higher power output for Bluetooth devices, thus increasing output power and range. Furthermore, devices in an environment can be strategically chosen and laid out such that it will minimize interference, such as using IEEE 802.11a devices, which do not operate on the 2.4GHz spectrum. A great amount of research is being conducted on Bluetooth-802.11 coexistence. These papers will further serve to address any interference issues in the future.

There is also an issue of connection and search times. To query CORE with a NameSpecifier, the INS tree lookup is very fast, probably O(n*logm), where n is the number of different entries in the tree (i.e. the number of root nodes) and m is the depth of the tree (which is usually on the order of 10^1). To perform an inquiry of local Bluetooth devices, the average time is approximately 2.56 seconds (see section 2.3.5), which is orders of magnitude greater than an internal CORE lookup. A connection to a Bluetooth device also takes an average of 0.64 seconds, and this may be a concern in the future as well.

The advantages, however, outweigh the challenges. Many CORE applications may not require high-bandwidth from nodes, and the packet and payload limitations may not be a problem. As CORE's serialization and deserialization becomes more efficient, overhead in CORE messages will reduce and we can utilize most of the Bluetooth packet payload space. In addition, whereas Bluetooth devices in a piconet communicated point-to-point, Bluetooth over CORE can communicate in an anycast and multicast fashion in addition to point-to-point.

CORE can but is not restricted to use wireless devices. Therefore, problems of interference can be minimized by not deploying wireless devices in the same local area as Bluetooth devices if operating on the 2.4GHz spectrum. As mentioned above, the issue with interference has many solutions, and further research will further serve to alleviate this concern.

Concerns with connection delays may also be minimal. In an environment with fairly stable and static Bluetooth devices, BluetoothAgent can cache Bluetooth addresses and service descriptions of local devices. This will allow for significantly faster connection times when queried by CORE. More importantly, as we briefly discussed before, dynamic linking within CORE will allow Bluetooth devices to link, connect, and use Bluetooth and non-Bluetooth devices already connected to CORE with virtually no connection delays.

This chapter presented a design and reference implementation of an extension to CORE that allows Bluetooth to communicate over CORE; however, we ran into many challenges in the process. In the world of research, many times we are faced with the trouble and bugs with beta software and technologies, and this is also the case for CORE and Bluetooth as well. From beta software to incompatibilities with kernel versions and non-standardized APIs, there are still many issues to overcome before we can see this Bluetooth extension realized. But from the reference implementation we see promise, potential, and hope. CORE's abstraction and indirection allows CORE nodes to communicate and control Bluetooth devices. Dynamic linking within CORE also can solve some Bluetooth issues, such as the issue with connection times as discussed above. Overall, the routing mechanism CORE provides allows us to extend

Bluetooth beyond its normal capabilities and address some issues with the current specifications.

Chapter 4

Future Work

As mentioned before, CORE is a robust and extensible system. However, more work can and should be done with CORE that continues to show extensibility, scalability, and robustness. The rest of this chapter outlines brief descriptions for future extensions of CORE.

4.1 Reversibility

One of the main goals down the road for CORE is reversibility. This can be characterized in two forms. The first is a system-wide reverse mode in which the CORE "world" stops and things go back one step at a time. Implementing this is fairly straightforward as we would only have to keep track of messages and rules that fire. We already have the infrastructure and handle to code this since the RuleManager and MessageRouter classes have handles to each other. The second type of reversibility is one that is localized. Going reverse in this case would depend on nodes and the dependencies which come from them. This, however, is much more difficult to put together since message systems can become very intricate and entangled as we scale more nodes into the system. The following are possible models for reversibility:

- request/reply model (acknowledgements stored)
- transaction model (begin/end)

4.2 Visualization

One type of administration and debugging tool that would be useful is a visualization or GUI tool to see what is going on within CORE. This can just be a module that uses the listeners in NodeManager, LinkManager, and Router. When links, nodes, and messages pass though CORE, this module, which sits on the same machine with CORE, can send information to a local GUI or a remote GUI.

4.3 Non-Java-Specific Serialization and Deserialization

Currently, we use a Java package JSX, which is the Java Serialization package for XML [38]. This performs serialization into XML-like text strings. However, because this is a Java package, non-Java devices, services, and applications cannot send messages through CORE.

The fix and extension is quite simple. JSX was an easy solution because it was a simple plug-in module. However, we started two methods in the Link, Node, Rule, and Message classes that are meant for standardized XML message serialization and deserialization; these methods are `toString()` and `fromString(String)`. These methods basically describe the above four classes in an XML structure and output to text. Furthermore, the extension must incorporate these with the `serializeAndWrapMessage(Message)` method in the Message class to serialize the message. Then, the `fromString(String)` method in the Message class must be modified to reconstruct the objects by combining the necessary `fromString(String)` methods from other classes. In addition, a length attribute in the XML tags for elements will help the parsing during deserialization. Since the `toString()` methods no longer rely on JSX, any device, service, or application, Java or non-Java, can create these XML-based messages and send them over TCP/IP to communicate

with CORE. The source code contains more specific details about serialization and deserialization.

4.4 Security

Our current version of CORE does not have any security implemented. This means that any node can add itself, remove itself, add other nodes, remove other nodes, add links, remove links, and send and receive messages at will to and from CORE. This, of course, is not a very desirable characteristic for a large system. Adding security, such as access control lists and encryption, should not be very difficult if done in an end-to-end manner (i.e. by modifying the Agent and Router classes).

4.5 Connecting two CORE systems

We can abstract CORE even further by providing a means for one CORE to connect to another CORE as if it is just another node. The following steps and modifications may need to be made:

- Modify Router class to accept connections with multiple Uids
- Add command to connect with a "CORE"
- Broadcast service announcements of CORE nodes
- Use sendQuery command to query a CORE
- Create a Network Address Translation (NAT) or Address Resolution Protocol (ARP) for Uid to Uid, or IP to Uid, etc

4.6 RES

RES is the remote execution server. The job of this module is to remotely manage the starting and stopping of nodes in CORE. One use for RES is when there are multiple services requiring the same resource. An extension of RES is one possibility for resource management in the future of CORE.

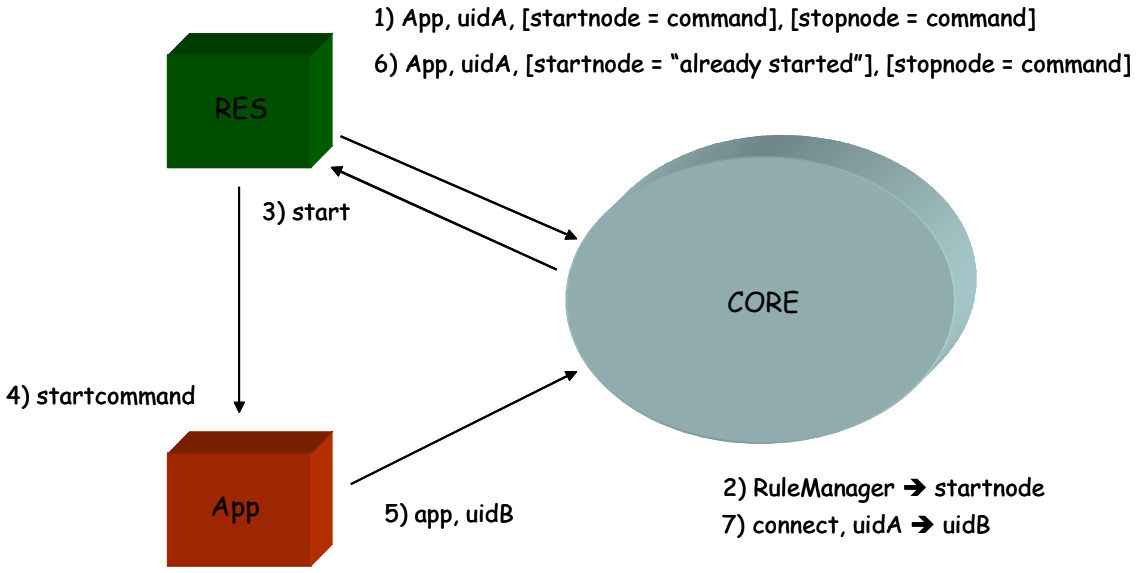

Figure 14 - CORE, RES, Application communication cycle

Figure 14 demonstrates the sequence of actions for RES. First, when an application sends its NameSpecifier to RES, RES sends a NameSpecifier announcement to CORE with a start and stop command. These commands tell CORE what to use to start and stop the application through RES. When the rule manager receives a request to start this particular application, CORE finds this start command from the node manager and sends this command to RES. Next, RES contacts the application with the start command and the IP address of CORE. The application then contacts CORE directly, and RES follows with another NameSpecifier

announcement; however, the NameSpecifer contains an attribute-value pair that tells CORE the application is already started. From then on, other nodes can connect and communicate with this newly-started application directly through CORE. Stopping a node through RES has a similar sequence of events.

4.7 Selective Rule Processing

Currently, rules are processed exhaustively, meaning that all rules are checked independent of how the state of CORE has changed. This presents much inefficiency in large systems with huge rule bases. A possible extension and modification to CORE would be to have a rule manager that does not process rules exhaustively; only process rules that are relevant to the latest state change of the system. Take the example of an agent addition; CORE can locate only the rules that have to do with agent additions and process those only. This selective rule processing can be done on a general or more specific level depending on the state change and thus saving significant rule processing time in large systems containing many rules and allowing for simpler scalability.

In terms of implementation, this may involve putting in an extra <rule-type> field and either creating a separate rule base for each rule type or checking preconditions only if the rule-type needs to be checked. Inserting the extra field is fairly simple since there are currently only four rule types.

4.8 Resource Discovery

Currently, RES is used to start and stop nodes in CORE. However, if a type of node does not exist in CORE, we would want something that can possibly go out and search for a suitable application or service. This is similar to the Microsoft .NET web-service paradigm. In this

paradigm, there exists a "yellow pages" of services that one can access on the Internet. These services can be found based on attributes, similar to how CORE uses NameSpecifier objects.

An extension to CORE in this area would mean implementing a way to interface with the web service directories. In addition, this may involve automatic generation wrappers for these web services to communicate with CORE.

Other options would be to integrate/implement JINI or some other third-party discovery protocol. As previously mentioned incorporating Bluetooth with CORE will provide another set of devices and expose an existing service discovery protocol as well.

4.9 Resource Management

Resource management is much needed addition for CORE. Currently, the rule manager does not allow links to nodes that already are involved in links. However, certain applications or services can accept multiple connections. Another scenario is if all the projectors in a building are being used, then the resource management module can process and reallocate resources based on priority. This is similar to thread-handling in Java and resource management for operating systems.

As discussed in section 2.1.4, Rascal is a resource management tool for Metaglue. Since it is written in Java, it is possible to develop a wrapper that will allow integration of Rascal and CORE.

4.10 Implementation of BluetoothAgent and BluetoothClient

Although Bluetooth has been on the market for quite a while, the move to standardize Java APIs for Bluetooth, or JSR-82, did not pass until April 2002. This meant that communicating

directly with Bluetooth devices via Java was not possible but could be achieved at a lower level, such as with COM or C. Still determined to find a viable solution, we searched to find a pure Java implementation of the Bluetooth stack and APIs to access them.

One suggestion for possible implementation is the beta developer kit from Rococco Software, Ltd., called Impronto DK and Impronto Simulator [26]. This beta version contained a reference implementation of JSR-82 for J2ME as well as a Bluetooth stack and simulator. However, this kit was written for a specific version of Redhat Linux and a specific kernel version. After a couple weeks of attempting to get the implementation working, we decided to search for alternatives.

We also suggest a JSR-82 implementation and Bluetooth stack from Atinav Inc [20]. They offer free downloads of their products; however, it is not a direct download from their site. After about eight weeks of correspondence, we were finally able to obtain their J2SE implementation of JSR-82 and Java Bluetooth stack. Unfortunately, after testing and debugging for about a week, we were unable to use this product to communicate with our Bluetooth hardware.

The full release of the Impronto developer kit and simulator is due out in the summer of 2002 priced at $2,500 and $1,000, respectively. Similarly, Zucutto Wireless Inc. produced their own reference implementation and Bluetooth stack, termed XJB100, for a whiteboard application; they sell the reference implementation and stack for $2,995. Discounts for research and academic purposes are minimal; however, as time progresses, there should be more open source Java-Bluetooth projects available on sites such as SourceForge [27].

The next steps of completing the implementation include finding a working JSR-82 implementation and Bluetooth stack and completing and testing the BluetoothAgent and

BluetoothClient implementation. There are a sufficient number of comments in the code to guide future researchers to test and debug the implementation. Furthermore, they will be responsible for integration with CORE and adding the service discovery module, if desired.

We have provided a framework and reference implementation for implementing this module. As developing standards converge for the Bluetooth arena, this extension will be possible to implement and incorporate into CORE.

4.11 Wireless Home Scenario

The Bluetooth home networking scenario mentioned above is closer to our homes than we realize. Current wireless technologies will allow users to move away from the traditional plug-and-play model to a discovery-based, context-aware input-output model. Future research will attempt to address this approach by implementing a system that can support this model.

Since there are few proprietary Bluetooth input devices on the market yet, it may be necessary to develop a simple device to make devices Bluetooth-compatible, such as for communication between a Bluetooth-enabled mouse to communicate with a laptop or Compaq IPAQ. There are devices in developmental stages, such as a Bluetooth-enabled virtual keyboard or a Bluetooth mouse for laptops.

Besides Bluetooth-enabled input devices, it is also necessary to acquire multiple output devices that are also Bluetooth-compatible, such as printers, an IPAQ from the 3800 series, laptops, or interface cards which will make the above devices Bluetooth-enabled. These exist in PCMCIA or Compact Flash formats. The latest Bluetooth devices can be found on the Internet [25].

It may or may not be necessary to acquire a Bluetooth access point. An access point will allow Bluetooth devices to access the local area network, for instance. However, the better approach may be to form ad-hoc networks, called piconets and scatternets. Many non-trivial Bluetooth wireless environments have been created and used recently, so ad-hoc networks is a viable solution.

The BluetoothAgent implementation is a required to achieve this scenario. With CORE, we have the ability to enable point-to-point or point-to-multipoint communication. For instance, the Bluetooth-enabled keyboard may broadcast in a push system to all devices that need keyboard input. Next, the rules within CORE will determine which connection(s) to maintain based on the context of the keyboard or input to the keyboard. These rules may be the most difficult part of this scenario; there have been numerous research projects on context and semantic understanding. The other approach is to write another module, such as our WinampSpeechAgent, and have all input devices send input to this module for processing. We then extend CORE to include dynamic linking, such as in the CORE VNC application [18], to be able to send data to the appropriate devices connected to CORE.

Chapter 5

Conclusion

We see that Bluetooth in a pure peer-to-peer network is not very helpful by itself. With CORE, we are able to integrate Bluetooth devices and services, giving Bluetooth devices the ability to use and be used by non-Bluetooth devices. This potential addition of Bluetooth to CORE is tremendous. Since CORE can run possibly anywhere in the world, this means that Java applications anywhere in the world can communicate and use any Bluetooth device anywhere in the world, given that it can connect to CORE.

Today, we find that computing power is increasing and being added to smaller, and more common devices. In addition, networking capabilities exist in a majority of these devices. Bluetooth, for instance, is an inexpensive technology and targets devices with power constraints, allowing networking abilities to be more ubiquitous. By combining and pooling computing power and resources of everyday devices, we find ourselves in the paradigm of pervasive computing.

CORE takes existing standards and technologies to extend us into a new paradigm of routing and service discovery. It leverages TCP/IP to enable non-specific devices, services, and applications to be able to communicate with one another. Our research in this area shows the potential of re-thinking and re-analyzing current routing, service discovery, and client/server paradigms.

References

[1] K. Takashio, M. Mori, M. Funayama, H. Tokuda, "Constructing Environment-Aware Mobile Applications Adaptive to Small, Networked Appliances in Pervasive Computing Environment," 2001.

[2] G. Abowd, "Ubiquitous Computing: Research Themes and Open Issues From an Applications Perspective," *CHI `97*, 1997.

[3] L. Rudolph, "Presentation Manager," 6.894 Lecture 1, 2001.

[4] B. Myers, "Using Hand-Held Devices and PCs Together," *Communications of the ACM*, Vol. 44, Issue 11, pp. 34-41, November 2001.

[5] Huang, B. Ling, J. Barton, A. Fox, "Making Computers Disappear: Appliance Data Services", *MOBICOM 2001*, Rome, Italy, 2001.

[6] Phillips, "Metaglue: A Programming Language for Multi-Agent Systems," M.Eng Thesis, 1999.

[7] L. Rudolph, "Project Oxygen: pervasive, human-centric computing – an initial experience," preprint, 2001.

[8] A. Kaminsky, H. Bischof, J. Coles, B. Koponen, J. Myers, J. Rigby, http://www.cs.rit.edu/~anhinga/, 2001.

[9] L. Rudolph, "Presentation Manager," 6.894 Lecture 1, 2001.

[10] S. D. Gribble, M. Welsh, R. Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao, **"**The Ninja Architecture for Robust Internet-Scale Systems and Services," *Special Issue of Computer Networks on Pervasive Computing*, 2000.

[11] Alan Kaminsky. "Infrastructure for Distributed Applications in Ad Hoc Networks of Small Mobile Wireless Devices." May 22, 2001.

[12] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," *Proc. 17th ACM SOSP*, Kiawah Island, SC, Dec. 1999.

[13] Microsoft Corporation staff, "Understanding Universal Plug and Play," June 2000.

[14] G. Tan, A. Mui, J. Guttag, and H. Balakrishnan, "Forming Scatternets from Bluetooth Personal Networks," *MIT-LCS-TR-826*, October 2001.

[15] J. Eker, A. Cervin, and A. Horjel, "Distributed Wireless Control Using Bluetooth," *IFAC Conference on New Technologies for Computer Control*, Hong Kong, P.R.China, November2001.

[16] S. Avancha, A. Joshi, and T. Finin, "Enhancing the Bluetooth Service Discover Protocol," 2000.

[17] P. Dahlberg and J. Sanneblad, "The Use of Bluetooth Enabled PDAs," 2000.

[18] Brendan Kao, "Design and Implementation of a Generalized Device Interconnect," 2002.

[19] C. Franklin, "How Bluetooth Works," http://www.howstuffworks.com/bluetooth.htm, 2000.

[20] Atinav Corporation, Bluetooth APIs, http://www.atinav.com/download.htm, 2002.

[21] M. Berggren, "Wireless communication in telemedicine using Bluetooth and IEEE 802.11b," *Technical report 2001-028*, November 2001.

[22] Zucutto Wireless Inc., Functional Java Bluetooth Stack, http://www.zucotto.com/products/xjb100.html, 2002.

[23] JSR-82, Java APIs for Bluetooth, http://www.jcp.org/jsr/detail/82.jsp, 2002.

[24] J. Schuurmans and D. Baremans, "Online Bluetooth Thesis for Telematics," http://infolab.kub.nl/edu/telematica/scripties00/groep3/#, Tilburg University, 2002.

[25] Blueunplugged.com, Bluetooth Products Galore, http://www.blueunplugged.com/main.asp, 2002.

[26] Rococco Software Ltd., Java APIs for Bluetooth, http://www.rococosoft.com/, 2002.

[27] SourceForge, Open Source Projects, http://sourceforge.net/, 2002.

[28] Sun Microsystems, RMI Technology 1.3.1 API Documentation, http://java.sun.com/j2se/1.3/docs/api/java/rmi/package-summary.html, 2002.

[29] Sun Microsystems, JINI Network Technology Architectural Overview, http://www.sun.com/jini/whitepapers/architecture.html, 2002.

[30] C. Dabrowski and K. Mills, "Analyzing Properties and Behavior of Service Discovery Protocols using an Architecture-based Approach," *Draft Copy*, June 2001.

[31] Sun Microsystems, JavaSpaces Technology, http://java.sun.com/products/javaspaces/, 2002.

[32] Sun Microsystems, "JavaSpaces Service Specifications Version 1.1", pg 4, October 2000.

[33] R. E. McGrath and M. D. Mickunas, "An Object-Oriented Framework for Smart Spaces," *Draft Copy*, 2001.

[34] Krzystof Gajos, "Rascal - a Resource Manager for Multi Agent System in Smart Spaces," "Proceedings of The Second International Workshop of Central and Eastern Europe on Multi-Agent Systems CEEMAS," 2001.

[35] Sun Microsystems, Jini Technology 1.2.1 API Documentation, http://java.sun.com/products/jini/1.2.1/docs/api/, 2002.

[36] Jeremy Lilley, "Scalability in an Intentional Naming System," Massachusetts Institute of Technology, 2000.

[37] Matt Welsh, NBIO: Nonblocking I/O for Java, http://www.cs.berkeley.edu/~mdw/proj/java-nbio/, April 2002.

[38] JSX Technical Homepage, http://www.csse.monash.edu.au/~bren/JSX/tech.html, 2002.

[39] SLS, SpeechBuilder, http://www.sls.lcs.mit.edu/sls/technologies/speechbuilder.html, 2002.

Appendix A: CORE

Indeed there are many different devices today with different platforms, protocols, and interfaces. However, we also know that computing power and memory are increasing and prices decreasing at an incredible rate. This translates into computing power all around us wherever we are and wherever we go; computing power as free and available as oxygen. CORE takes an attempt at combining devices and pooling resources to create an environment where different devices with different protocols can communicate with the minimum of requirements.

Figure 15 depicts applications and services running possibly different operating systems and form factors connecting to one CORE system. They are able to communicate with each other and share resources despite the differences between them. This is this objective that the CORE system seeks to accomplish. CORE is a generalized smart decentralized routing system, and it is important to discuss design considerations and implementation details to show the extensibility of CORE. This chapter describes some design considerations, basic modules, and some implementation details of CORE.

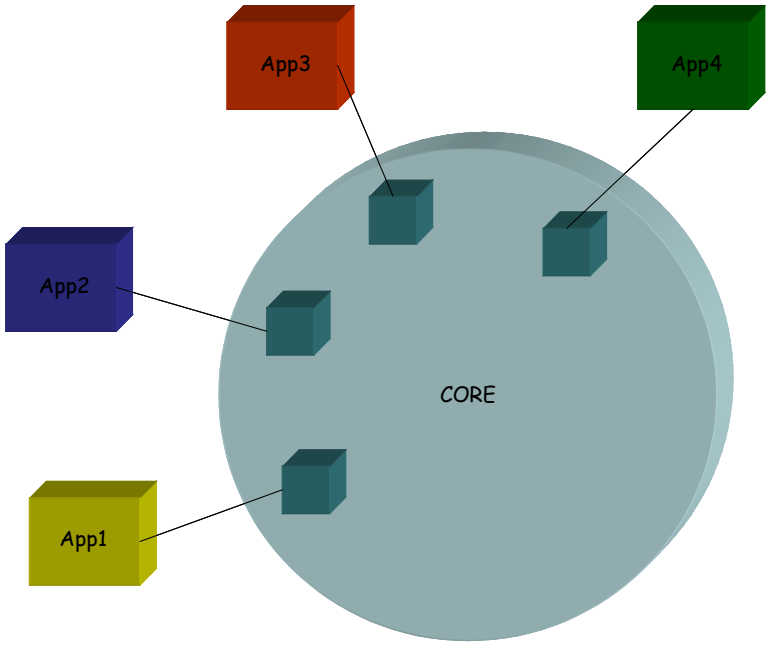

Figure 15 - CORE and Applications/Services

A.1 Scenarios

A generalized communication-based router would provide great functionality and usefulness in the world of pervasive computing. Not only this, CORE is also extensible to add resource management as well as many other possible features. The following few scenarios show how CORE can be applied to home and corporate use.

A.1.1 Presentation Manager

Dr. Larry Rudolph, a principle research scientist at the Massachusetts Institute of Technology, along with his research team designed and implemented a presentation manager based on an earlier version of CORE (see Figure 16). The main goal of the Presentation Manager is to create a human-centric environment in which people can give presentations without concerning himself about certain logistics. Imagine the following:

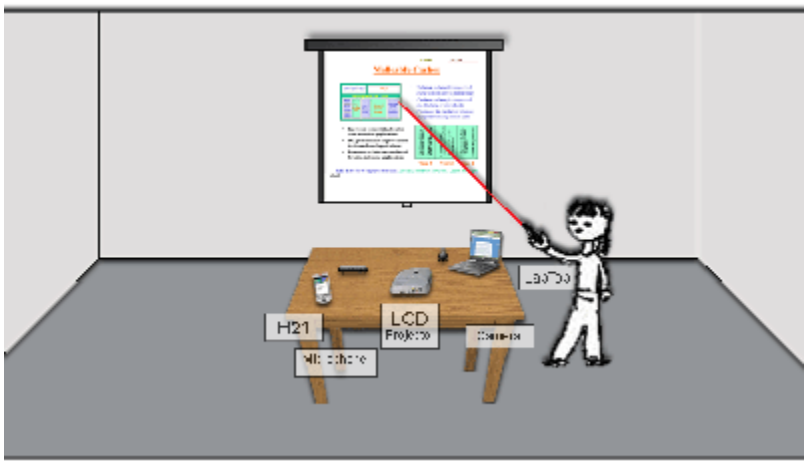

Figure 16 - Presentation Manager

We are giving a presentation today. Our laptop crashes and we forgot your note cards as well! Well, all this is not a problem since the Presentation Manager will be able to help us out as long as your presentation is stored on the LAN or accessible over the Internet. First, as we walk in to the room, the Presentation Manager will use face recognition, voice recognition, or some absolute pointer to pull up your presentation based on factors, such as time, meeting topic, etc. Next, your presentation will display on the main projector and editable copies will display on the laptops connected to CORE in the meeting room. Following that, localized copies of your presentation will be sent to a handheld computer and laptop in the front of the room. In addition, the presentation will can either be controlled by speech or with the traditional keyboard input.

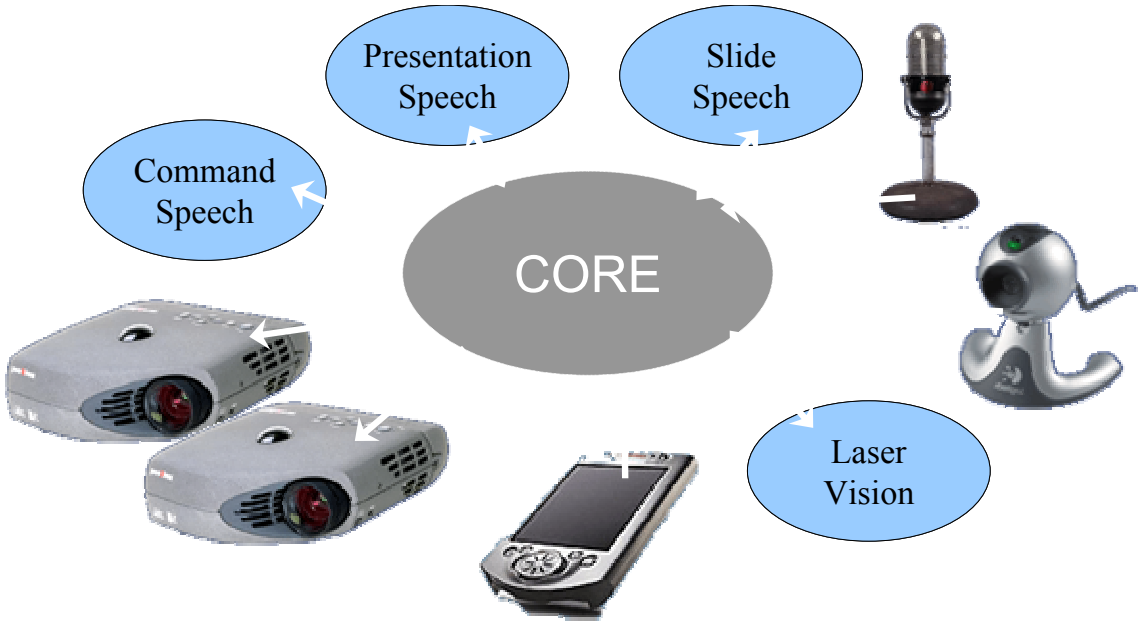

Figure 17 - Presentation Manager through CORE

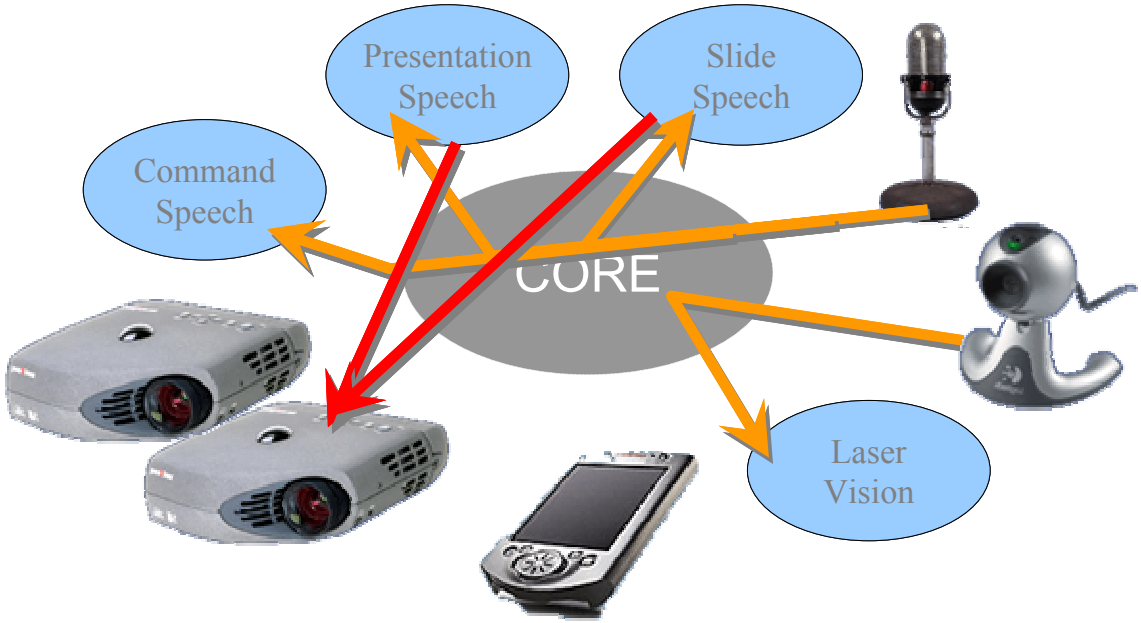

Figure 18 - Links from/to Nodes in CORE

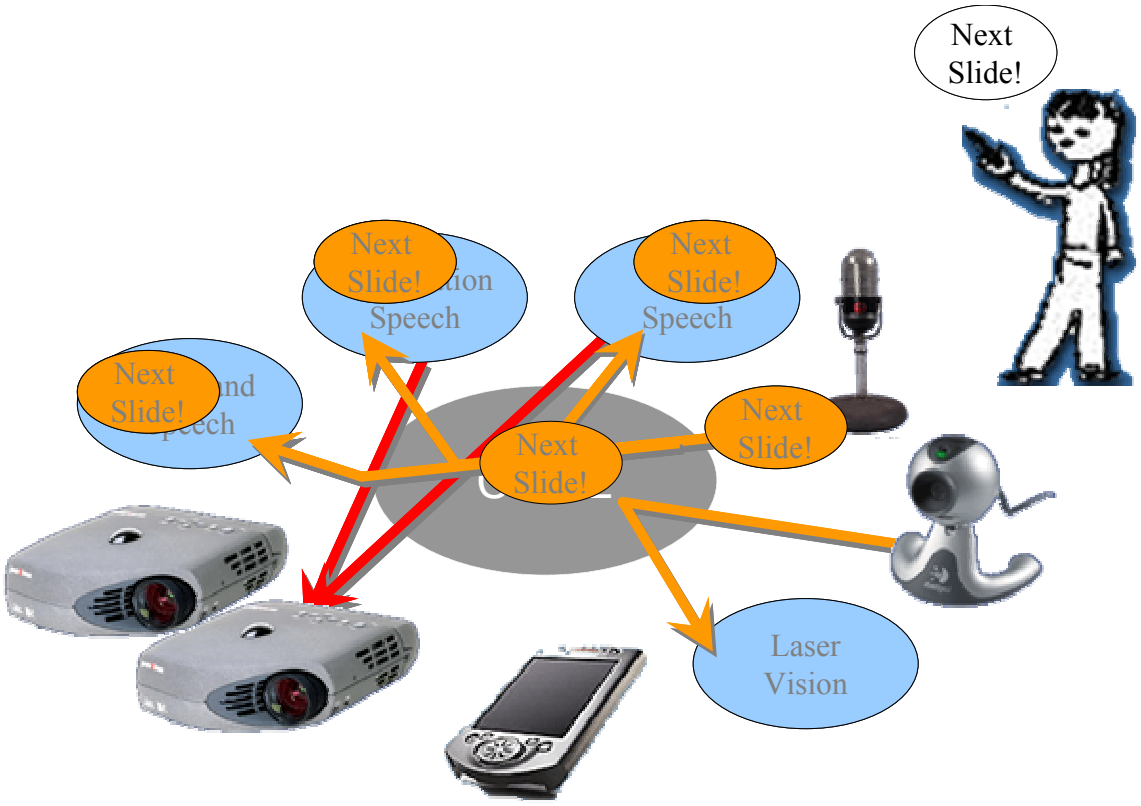

Figure 19 - "Next Slide" Command Input Through CORE

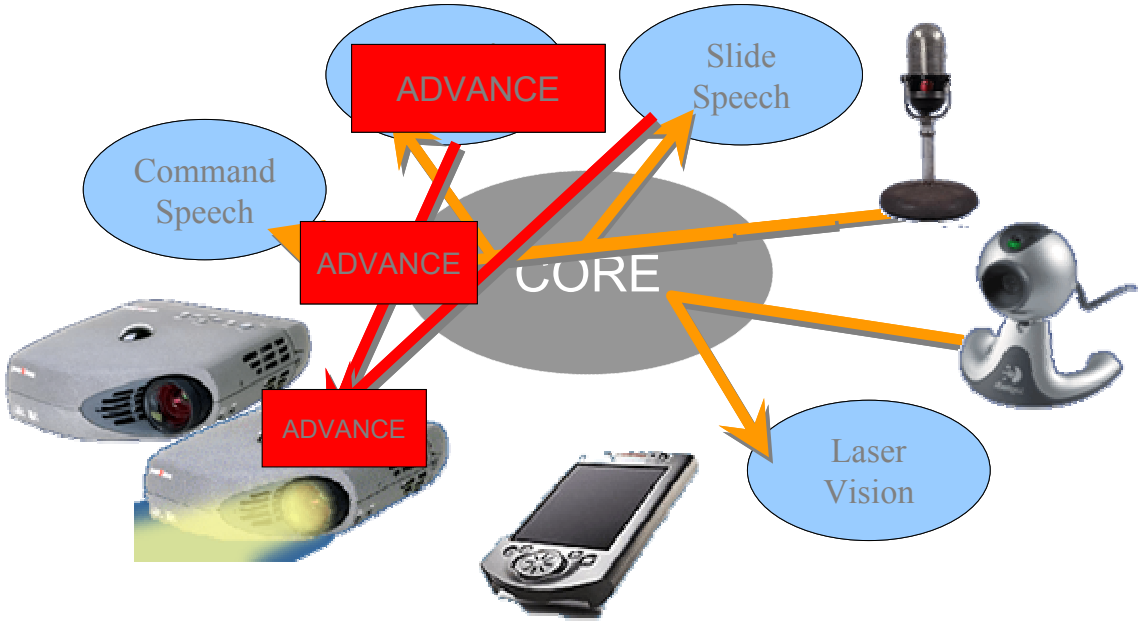

Figure 20 - Advance Command Sent through CORE

Figure 17 shows CORE and the nodes attached to CORE. The nodes in clockwise order are: a speech command module, a presentation speech module, a presentation application, a speech an input device, a firewire camera, a laser pointer tracking application, and two projectors.

Figure 18 shows uni-directional links in CORE.

Figure 19 shows a human's speech input ("next slide") being sent over CORE to appropriate destinations governed by the links established prior to speech input.

After the speech is processed, the speech processors output and send appropriate commands. Figure 20 shows the advancement of the slide presentation. This could happen in two ways. The application processes the speech and sends the command directly to the presentation application, or a rule can trigger, and CORE sends the appropriate commands based on the consequence of the rule fired.

A.2 Design Considerations

The goal of CORE is to create a system that is human-centric in its use and control. In other words, the CORE system allows the user to perform simple as well as complicated computational tasks using a variety of different non-homologous hardware and software agents together without having to understand the lower level connectivity issues that arise from linking devices and services that were not created to work together automatically. In the process of designing CORE, we ran into many issues that needed consideration and documentation. The rest of this section discusses a few of these considerations.

A.2.1 TCP and NBIO

The first version of CORE was built in Java and all the network communication was done over TCP/IP. This combination provides a unique problem, as JDK 1.3, the newest version

of the JDK available at the beginning of development, did not support certain non-blocking operations (i.e. non-blocking read) on TCP sockets including writing and accepting connections. This means that agents and CORE, itself, would require threads for each connection – an unfriendly development task. INS, also implemented in Java, avoids this problem by using UDP instead. Metaglue does not have this problem as it uses RMI, which handles all the actual network connections. We felt using UDP and adding acknowledgments would be reinventing TCP and did not justify the added amounts of code. The solution was to use Non-Blocking I/O (NBIO) a third party package that adds non-blocking sockets to Java [37]. This package has henceforth been integrated with the JDK, first available in JDK v1.4.

A.2.2 Java 1.4 "NBIO" vs. NBIO

With the recent release of JDK v1.4 as this thesis was being written, the many new features came with many great bugs. Our decision to forgo using a standardized API in favor of NBIO is mainly for stability concerns. One tradeoff that follows is that NBIO is written for the Linux OS, and the more recent Windows 2000 version is still in the beta stage. This tradeoff means that CORE does not meet the "platform-independent" promise completely; however, Matt Welsh, designing and creator of NBIO, claims porting from NBIO to Java 1.4 is not too difficult, and at the time this was written Welsh was in the process of converting his project code over to the Java 1.4 API.

A.2.3 INS naming

We chose to use the INS to describe devices and services. This is mainly because most of the functionality we require has already been implemented in this naming scheme. In addition, if we require INS for other purposes in the feature, then the use of its own naming scheme will facilitate integration more easily.

A.3 Basic Components

The CORE system is composed of four main components: nodes, links, messages, and rules.

A.3.1 Nodes

Nodes are points of attachment where a hardware device or an enduring software service is wrapped with the appropriate formats and connected to the system. Nodes can be attached to CORE, started, stopped, and removed from CORE. The CORE system then automatically reconfigures to adapt to these changes.

The actual Node object is nothing more than an INS name specifier with a UID. However, the CORE representation of example nodes can be symbolized with the following metadata format:

- *nhardware0 :* ["Device", "device_name", "Location", "location_name"]
- *nsoftware0 :* ["Process", "service_name", "OS", "OS_name", "Owner", "owner_name"]
- *nsoftware1 :* ["Process", "service_name", "Platform", "platform_name", "Owner", "owner_name"]

A.3.1.1 Node Design Goals

Nodes are the building blocks of the system. They are the individual units that are connected together to make the system produce functional usefulness and flexibility. There are several goals we must keep in mind when designing the nodes.

- Nodes must be interface-independent. This allows for agents and services of all types to be attached to our system.

- Nodes must be able to be dynamically attached, started, stopped, and detached without affecting the flow or functionality of the system.
- Nodes need smart naming scheme so resources can be discovered based on node properties and location.
- Nodes need state roll back capabilities (future extension).

A.3.1.2 Node Manager

The node manger is responsible for all the node specific activities that involve attaching, starting, stopping, and detaching. It maintains a database of all available nodes and their current states. In addition, the node manager acts as a "garbage collector" in the sense that it periodically checks for "dead" nodes by requiring nodes to announce its existence within a certain refresh period.

A.3.1.3 Agent Naming

Agent naming is done through INS syntax and are based on the agent's location, intention, and any other agent characteristics needed to form an entry in the node resource table to be used for future discovery. In addition, some proposed mandatory attributes include:

- *core*=true
- *res*={true/false}
- *startcommand*={a command to tell RES how to start, if applicable}
- *stopcommand*={a command to tell RES how to stop, if applicable}
- *status*={running/stopped}
- *uuid*={INS-uuid}

"*res*" specifies whether or not the node is supported by RES (Remote Execution Server)[5]. If so, "*startcommand*" contains the command to send the RES to the service. And likewise, the "*stopcommand*" is for stopping the service. "*status*" indicates whether the service is currently running or not. "*uuid*" is used to uniquely identify an agent. Specifically, uuid is a 64 bit string generated from the MAC address and current time of the system.

A.3.1.4 Agent Initialization

When an agent requests to be added to the system as a node it must be able to contact the CORE using an available wrapper that broadcasts the agents location, intention, and other device or service characteristics (i.e. via an INS name-specifier). The agent then sends an announcement message that declares all relevant and necessary device characteristics for device naming and discovery purposes. These characteristics are processed by the Node Manager and feed into its Node tables for storage and future discovery purposes. See Agent description in Appendix B for more details.

A.3.1.5 Agent Destruction

When an agent wishes to leave a system gracefully, it notifies the node manager with a <remove> message. The server responds by determining all nodes that will be affected and rules that will be fired and performing the corresponding actions. In addition, the node manager notifies all listeners (in this implementation, only the link manager is listening) of a node being destroyed.

CORE can also handle ungraceful agent departures. The Node Manager sets a timeout for each agent, within which an agent (node) tells CORE that it is still running. After that time

[5] Some of these attributes are for the future RES extension.

runs out, the agent is considered dead. At this point CORE concludes that the agent has left based on a lack of response and performs the same routine as a graceful departure.

A.3.1.6 Agent Discovery

Agents are discovered by querying the CORE node manager. Agents wishing to find a specific agent or a type of agent can query the node manager with a Uid or a INS name specifier. The node manager responds with the status corresponding agent or a vector of potential Uids, respectively, to originating agent.

A.3.2 Links

Links represent connections between agents. Links in CORE are uni-directional, meaning that each link is a one-way link. Bi-directional links can be simulated with two uni-directional links. Links are important because CORE is a message-based routing system. Messages have no destinations specified. Instead, messages coming from a node in CORE are routed to *all* nodes that the sending node is linked to.

A.3.2.1 Link

A link is simply an object with a source and a destination. For creating rules, one may use name specifiers to link nodes to one or many nodes matching the name specifier.

A.3.2.2 Link Manager

The link manager stores all links in CORE within a hash table. The keys of the hash table are source Uids, and the values of the hash table are Vector objects holding Uids of destinations linked to the particular source.

A.3.3 Messages

Messages are the most vital part to the CORE system. In addition to using messages to add rules, nodes, and links, messages determine when and which rules fires and are also the main method of communication between nodes in CORE.

Messages are self-describing data units that are routed by CORE to the appropriate subsystems or other nodes. Messages contain headers that describe its type, protection scheme (future extension), source, and data length. Messages are moved through the system by CORE, which sets off different actions based on message type and content.

A.3.3.1 Message Types

There are currently six different types of messages within CORE:

- *RULE*: these messages add rules to the CORE rule base
- *CHECKED*: these messages are checked against the CORE rule base; these messages are automatically sent to all destination nodes linked to the sending node
- *RAW* (streaming/high-volume data): these messages are sent directly to all destination nodes linked to the sending node
- *ANNOUNCE*: these messages announce a node's description and Uid; these messages are used during agent initialization and agent refresh
- *QUERY*: these messages are used to query the CORE node manager for existence of specific nodes or nodes with queried name specifiers
- *RESPONSE*: these messages are *only* sent from CORE to response to nodes with information, such as an ACK, NACK, or responses to queries.

A.3.3.2 Message Format

Message format is not very important since it is mostly abstracted from the developers of CORE applications and services. The specs give developers the requirements for what CORE needs as a complete message. The following is the example of a complete message:

m_i : [<messageType>,<source>,<length>,<data>]

The java methods in the agent class require all of this content or a usable subset of this content; however, developers using the agent class never have to create a Message object directly (see Appendix B).

A.3.3.3 Message Content

This section describes the requirements or expectations for each message argument.

- <*messageType*> - these are static final integers in the Message class; either RULE, CHECKED, RAW, ANNOUNCE, or QUERY.
- <*encryption*> - future extension.
- <*source*> - this is a Uid string corresponding to the INS Uid of the service or application; this is automatically generated at the time of connection to CORE.
- <*length*> - this is the total integer length of the actual message data.
- <*data*> - this is the actual data passed in as an object and parsed based on the <*messageType*> - Rules are of the form of a Rule object; Checked messages are String objects; Raw data, such as streams, are sent as a byte array; Announcement messages are NameSpecifier objects; Query messages contain a Uid or a NameSpecifier object.

A.3.3.4 Sending Messages

Messages that are sent over TCP/IP are first serialized with a Java packaged called JSX. JSX is an XML-based string deserializer that is in accordance to Java's Serializable class. With JSX, serialization and deserialization can be achieved in a few lines of code. However, the downfall is that JSX serializes with a great deal of overhead; much of what is included in a simple object seems unnecessary for most uses. This may pose a challenge when we are dealing with applications or services with limited bandwidth. In future versions of CORE, we can create or substitute a different method of serialization and deserialization to increase efficiency. Since this module basically acts as a plug-in, changing serialization and deserialization methods is simple and straightforward.

A.3.3.5 Routing Messages

Messages are processed based on all the message type. Rules are routed directly to the Rule manager to be processed further. Checked messages are sent to the CORE rule manager to check against the rule base; these messages are also routed directly to the appropriate destinations. Raw data is not processed by CORE and is just routed to the nodes that are linked to the sending node; this data is usually audio/video streams and other high-volume data.

A.3.4 Rules

Rules are a very basic set of instructions that perform actions based on a current the state or propagating message. Rules are in the same form as those in a rule-based system: if the precondition statement is satisfied, then the rule is considered triggered, and if chosen to fire the consequence statement is carried out. In addition, there is a field in the Rule object that

tells CORE if the rule should be deleted after being fired, or if it should persist in the rule manager until another rule removes this from the rule base.

A.3.4.1 Rule Types

CORE has four basic object types: node, link, rule, and message. We decided that rules either have "+" or "-" as modifiers; this generally means "the existence of" or the "the absence of," respectively. The following a more detailed descriptions of the +/- modifiers for each object type in a precondition and consequence.

- "+ node" *pre*condition ➔ if: <node> exists in the CORE node manager and is *not* in use [i.e. not part of a link]
- "- node" *pre*condition ➔ if: <node> does not exist in CORE node manager, or all nodes of this type are in use [if NameSpecifier]
- "+ node" *post*condition ➔ use RES or resource discovery to start/connect a node
- "- node" *post*condition ➔ use RES to stop (*all* if NameSpecifier) a node (and disconnect *all* associated links)

- "+ link" *pre*condition ➔ if: any link exists between A ==> B or from a type of A to type of B (if NameSpecifier)
- "- link" *pre*condition ➔ if: no link exists between A ==> B or from a type of A to a type of B (if NameSpecifier)
- "+ link" *post*condition ➔ adds a link from A ==> B (A, B must already be started and B not in use [i.e. B is not already part of a link]) or from a type of A to a type of B (if NameSpecifier)

- "- link" *post*condition ➔ removes a link from A ==> B (does *not* stop the node) or a link from a type of A from a type of B (if NameSpecifier)
- "+ message" *pre*condition ➔ if: message is "currently" passing thru CORE (i.e. this message came through CORE during this round of rule processing)
- "- message" *pre*condition ➔ this makes *no* sense, so the CORE specifications do not allow this type of message
- "+ message" *post*condition ➔ send this message thru CORE as if it is coming from original source
- "- message" *post*condition ➔ this makes *no* sense, so the CORE specifications do not allow this type of message
- "+ rule" *pre*condition ➔ if: this rule exists currently in the CORE rule base
- "- rule" *pre*condition ➔ if: this rule does *not* exist in the CORE rule base
- "+ rule" *post*condition ➔ add this rule to the CORE rule base
- "- rule" *post*condition ➔ remove this rule from the CORE rule base

A.3.4.2 Rule Manager

Rules are stored in a vector and processed in a loop until there are no state changes from a complete loop through the rules. A state change can be events such as agent additions, agent removals, rule firings, actions, and any other event that causes the current state of the CORE server to change.

CORE was initially developed with a rule polling system that just continued to process rules without regard to state change. We found this inefficient, and since our design of CORE was

sufficiently robust and extensible it was not difficult to change the rule processing to a notification-based system. The main advantage here is efficiency, especially when we consider large-scale servers with thousands and thousands of rules and with hundreds of other threads needing processor time.

A.3.4.3 Rule Examples

The following are examples of some meta-rules. These are not what actual rules would look like since CORE requires either Uid or NameSpecifier objects for links and nodes.

- - monitor ➔ + monitor

This says that if there is no monitor-type device currently running, start one up and connect it to CORE if it is not already connected.

- - link(iPAQ ➔ monitor) ➔ + link(iPAQ ➔ monitor)

If there is no link connecting an iPAQ to a monitor, then create a link that connects the iPAQ to the monitor. (this is a general iPAQ, but we can specify location or even a specific iPAQ if needed)

- + iPAQ AND - monitor XX➔XX +monitor AND + link(iPAQ ➔ monitor)

Rules of this type are not possible. In CORE we can have compounded preconditions, but postconditions can only have one clause. We can break up the rule into the following two rules.

- + iPAQ AND – monitor ➔ + monitor

If there is an iPAQ (in the room) and no monitor (in the same room), then connect a monitor to CORE (in the same room as the iPAQ).

- - link (iPAQ ➔ monitor) ➔ + link(iPAQ ➔ monitor)

If there is an iPAQ and a monitor (in the same room) link the iPAQ to the monitor. This rule is a bit simplified since we may have to query CORE in case some other service or application is trying to access the same resources.

A.4 Implementation-Specific Details

When designing the system with the above goals in mind, these subsequent specifications outlined the path the system's design would take:

- *Agent independent.* Allows a general interface that all agents can communicate and interact with using wrappers/device drivers.
- *Dynamic addition of agent nodes.* Allows for the addition and removal of new and existing agents and their corresponding nodes.
- *Edges/Connections can be added and removed between agents.* Allows for the addition and removal of new edges between existing and new agent nodes.
- *Agent name based on agent location and purpose.* Allows for easy discovery of known components.
- *Rollback capabilities.* Allows for the rollback to previous states of the system. (future extension)
- *Secure messaging between agents.* Allows components that wish to speak with each other talk. (future extension)
- *Dynamic changing of system state based on rules and actions.* Allows for the system to dynamically change as new rules are added and actions taken.

- *Rules based on message content.* Allow for system to take appropriate action based on message content.

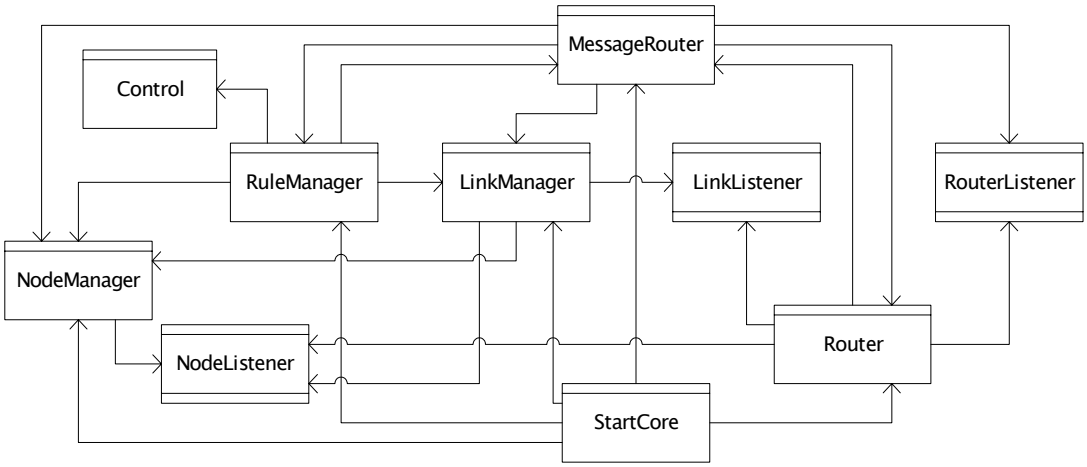

Figure 21 - Module Dependency Diagram (CORE)

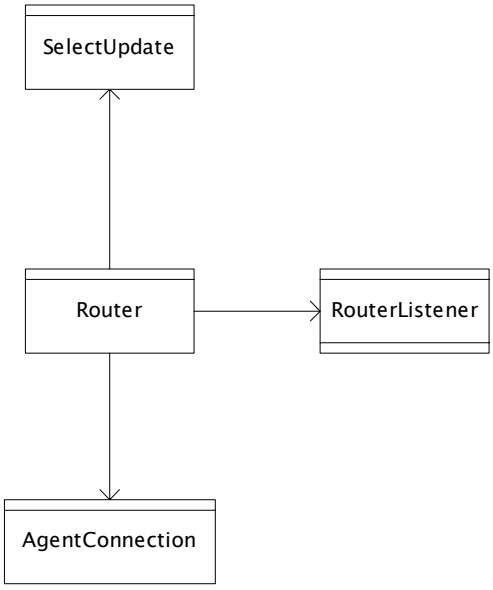

Figure 22 - Module Dependency Diagram (Router)

Figure 21 and Figure 22 show the module dependency diagrams for a CORE server (StartCore) and for Router.

A.4.1 Link Manager

There is nothing special about LinkManager. The data structure we chose to use is a hash table. The <key, value> pair represented is a <source Uid, Vector of destination Uids>. There are methods to add and remove links, as well as a method to check if a link exists in CORE. The LinkManager class also has an internal Listener class that other classes can implement to be notified when links are added or removed; in our version of CORE, RuleManager implements this for its notification-based rule processing.

A.4.2 Node Manager

NodeManager is to be CORE's resource management module. It adds nodes when it hears an announcement from CORE, and it has the ability to start and stop nodes. NodeManager stores NodeTree objects in a hash table. Other nodes can query NodeManager to check the status of a node. As with LinkManager, NodeManager has an internal Listener class to notify listeners that a node has been added, removed, started, or stopped. Finally, the NodeManager has an internal thread that performs a "garbage collection" routine that removes nodes that do not refresh announcements to CORE.

A.4.3 Rule Manager

The RuleManager class acts as the main logic behind CORE. In general, the rule manager is a another rule-based system; however, there are a few complications because of the generalized NameSpecifer objects.

First, we create the interfaces CoreCondition and CoreConsequence. The Link, Node, Rule, and Message objects implement these interfaces. In addition, there is a vector that acts as a message buffer for CORE *CHECKED* messages that come through; these messages are

removed from the buffer if and when a complete loop of the CORE rule base completes. RuleManager also implements the LinkManager and NodeManager interfaces, so that when the state of CORE changes, the rule processor is notified to process rules once again if it was waiting for a state change, or the rule processor continues looping through the rule base if it is currently processing rules.

While processing rules, RuleManager checks CoreCondition objects. If a precondition is satisfied, the rule processor fires the consequence. The complication is that when a NameSpecifier is part of the precondition, most likely the creator of the rule wants the same object returned from the query to be used in the consequence statement. Currently, the RuleManager can handle simple rules that need this feature; however, using NameSpecifier objects in preconditions in conjunction with AndRule objects (i.e. compounded rules), the RuleManager processor complains and warn that it may not return with the desired effect. One work-around for this was our implementation of the *QUERY* message type. A rule-writer can query with NameSpecifier objects and send rules with specific Uids instead of NameSpecifier objects.

A.4.4 Message Router

MessageRouter is one of the central classes to CORE. It has a handle to NodeManager, LinkManager, RuleManager, and Router since CORE is a message-based system. As mentioned above, there are currently six types of messages in CORE; however, it is fairly simple to add more message types to CORE. All messages coming in to CORE are parsed by MessageRouter and processed according to its message type.

A.4.5 Router

The Router serves as the abstraction between the physical network and the rest of CORE. The rest of CORE only needs two functionality of CORE: to be notified on the arrival of a message on the network and the ability to send a message to a specifying node by specify the node's UID. As mentioned previously, CORE uses TCP for the network connections between CORE and each of the nodes. Since we used a non-blocking I/O package to do this, there is no need for Router to be multi-threaded. The Router keeps track as each Node connects, and by watching the announcement messages, it associates specific TCP connections with the UID. The method that Router provides to the rest of CORE to send a message is non-blocking, so instead of blocking until a message is completely sent, Router just tacks the data on to the outgoing buffer of the proper TCP connection -- and sends the data whenever it can.

A.5 Running CORE

CORE is very straightforward to start up and run. The server contains a node manager, a link manager, a rule manager, a message router, and a router (see Figure 23). First, instantiate a NodeManager object. Then, instantiate a LinkManager with the NodeManager as the argument. Create a RuleManager with the NodeManager and LinkManager passed in as arguments. Start up a Router with the port we want it listening on (the default is port 10626). Next, instantiate a MessageRouter and pass in all of the above managers and the router. Next, have the CORE router add the MessageRouter as a listener, and have the RuleManager get access to the CORE router to be able to send messages. Finally, start the rule processing thread in the RuleManager and have the CORE router call its run method.

Once CORE is started up and running, it sits and waits on the port for messages. As mentioned above, messages can add rules, add messages, send raw data, send queries, or announce to CORE.

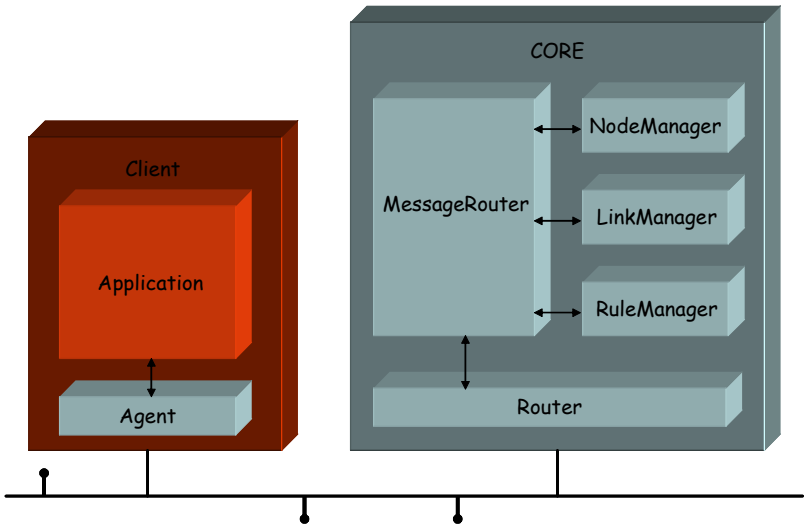

Figure 23 - CORE, Agent, Application on Network Layer

Appendix B: CORE Agent API & Applications

Appendix A described the server side of CORE. To allow clients (i.e. CORE applications and services) to communicate with CORE, we needed to create a proxy or client stub in which applications and services either extend or use directly. We chose the latter, which was to create a class that developers either extend or instantiate and call directly. The rest of this chapter describes the Agent class and a few example implementations with this class.

B.1 CORE Agent

The CORE agent allows applications and services to communicate with CORE. Once a connection is established with CORE, the agent serves CORE with the following functionality: sending data, receiving data, and announcing of its service.

B.1.1 Connecting to CORE with the Agent class

First, an application must instantiate an instance of Agent with a host and port. The *host* refers to the IP address of where CORE runs and the *port* is the port where CORE is listening on. Next, the application must specify a NameSpecifier for itself. For example, a Winamp application may have the name specifier:

```
[Device=Winamp] [core=true [res=false][uid=this.agent.getUid()] [status=stopped]]
```

Finally, the application calls the `connectAndRun()` method. From here on, the application can do whatever local processing it needs. To communicate with CORE, it can use methods from the following section.

B.1.2 Using Agent to Communicate with CORE

- `sendString(String)`

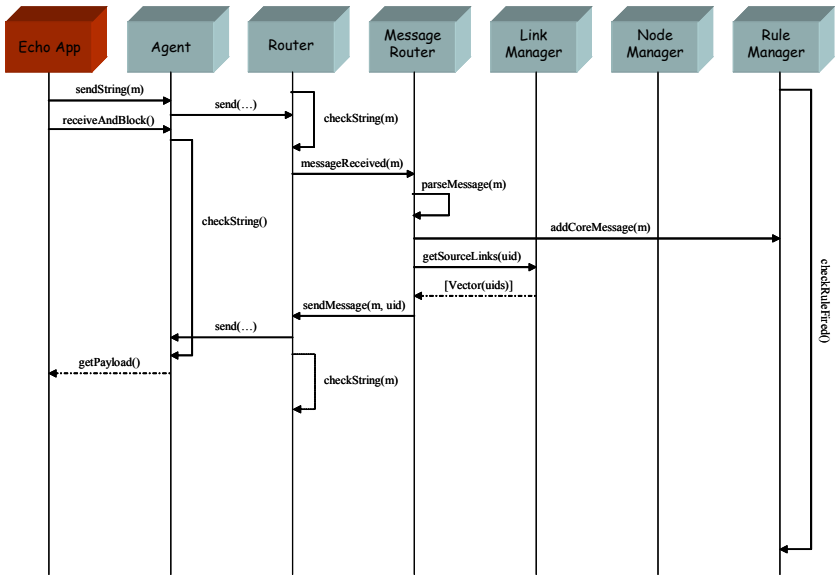

Figure 24 - Sequence Diagram (sendString)

Applications and services can use this method to send String objects across to CORE to be checked against the rule base as well as to be sent directly to all nodes connected to the sending node. For example, in the Presentation Manager scenario, an application can send PowerPoint a "next slide" string to tell the presentation to advance to the next slide. These messages may also trigger rules; in the above example, there may be a rule that tells CORE to update remote users when it sees a "next slide" command.

Figure 24 (above) shows the sequence diagram for `sendString(String)`.

- `sendRaw(byte[])`

Applications can also send raw data across CORE. For example, we have been working on a VNC client application that will receive desktop images over CORE. These images are

sent as video streams. In another application, one may want to send raw audio to a speaker connected to CORE. `sendRaw()` is meant for any high-volume data such as audio and video streams.

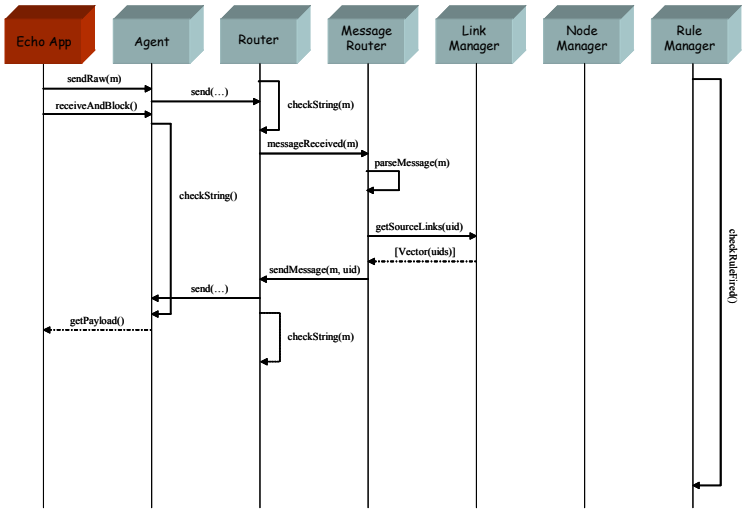

Figure 25 - Sequence Diagram (sendRaw)

Figure 25 (above) shows the sequence diagram for `sendRaw()`.

- `sendAnnounce()`

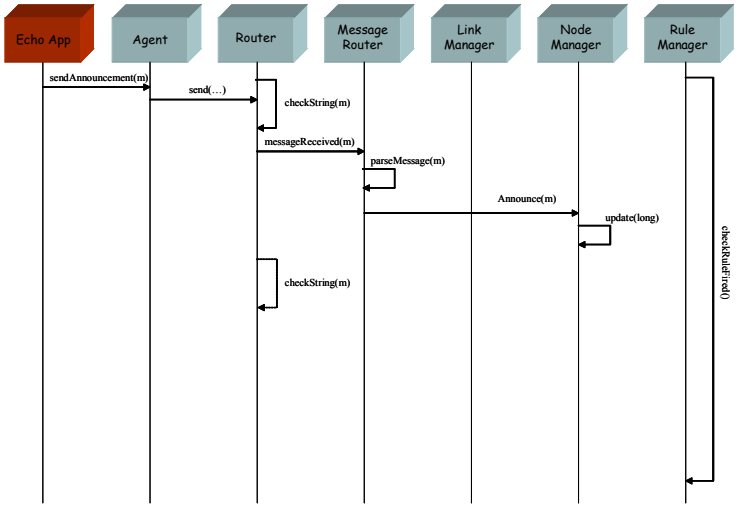

Figure 26 - Sequence Diagram (sendAnnounce)

All applications and services automatically send an announcement to CORE of its presence as long as it is running. The only requirement for this is that a NameSpecifier is set before an announcement message, and that the NameSpecifier forms to the specifications of CORE.

Figure 26 (above) shows the sequence diagram for `sendAnnounce()`.

- `sendRule(Rule)`

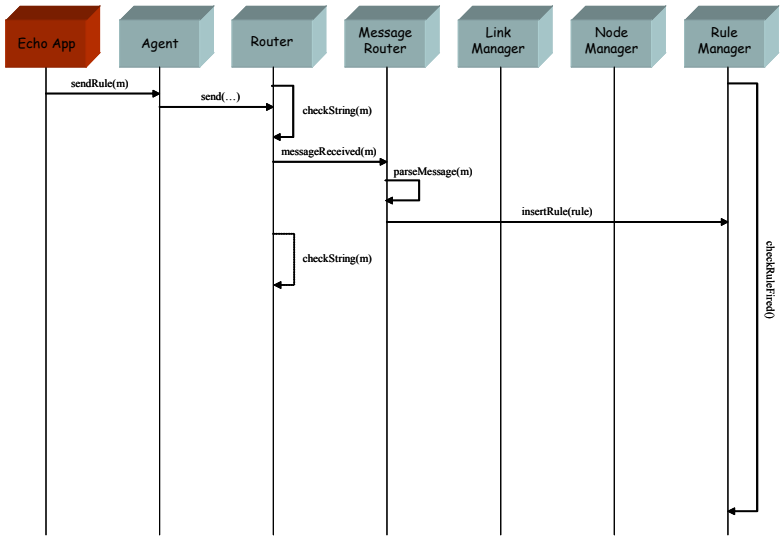

Figure 27 - Sequence Diagram (sendRule)

Applications desiring to insert rules into the CORE rule base must use this method. Rules are also the only way in which applications and services can create and destroy links and start and stop nodes. For example, an application wishing to create a link from itself to itself needs to write a rule that does so.

Figure 27 (above) shows the sequence diagram for `sendRule()`.

- `sendQuery(NameSpecifier/Uid)`

This late addition to CORE is important for applications and services wishing to connect to certain devices. Since CORE currently has no resource management module, this is the way in which applications and services can monitor the status of nodes or query types of nodes.

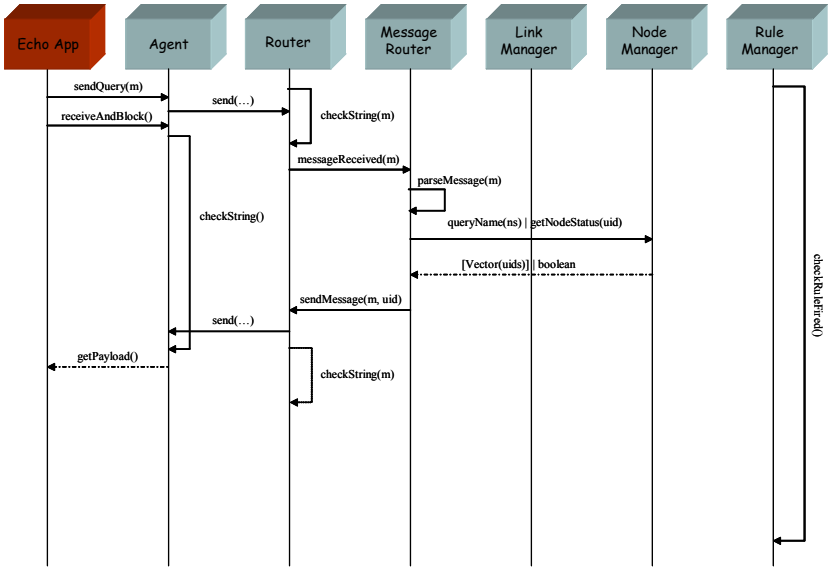

Figure 28 - Sequence Diagram (sendQuery)

`sendQuery()` has two possible methods of use. `sendQuery(NameSpecifier)` queries CORE to return a list of Uids that match the given NameSpecifier. `sendQuery(Uid)` queries CORE with a specific Uid, and CORE returns the status of the application or device (i.e. whether or not it is currently being used). For example, an application wanting to connect to a Winamp application but does not know where of if one exists can query CORE with the appropriate NameSpecifier. This returns a list of possible Uids of valid Winamp applications, and then the application can query CORE with the specific Uid to see if the Winamp application is currently being used. Finally, if it finds one that is not in use, it can send CORE a rule that tells CORE to connect the application to the Winamp application.

Figure 28 (above) shows the sequence diagram for `sendQuery()`.

- `receive() & receiveAndBlock()`

Agent has two methods for applications and services to receive data. `receive()` checks to see if there is any data on the receive buffer and then go on without blocking. `receiveAndBlock()` sits and waits (i.e. blocks) until it receives one whole response message.

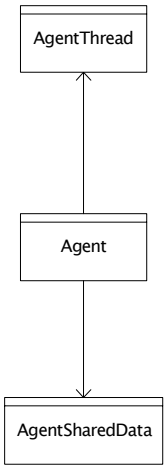

Figure 29 - Module Dependency Diagram (Agent)

Figure 29 (above) shows the MDD for Agent. The two above methods in Agent rely on its helper classes to receive and retrieve data for the application.

B.2 Example Applications

In order to test CORE and present some of its raw potential, we wrote a few small applications to show the ease of development and possible functionality. Most of these applications were developed in less than a few hours. The difficulty did not come with figuring out how to talk to CORE but with how to use C to talk with Winamp, for instance, or to figure out once again how to use SpeechBuilder, which is a speech processor written by researchers in the SLS group at MIT.

B.2.1 CommandWindowEchoApp

This application allows us to do three main things. The first one is to connect links to other nodes running on CORE. The second function is to destroy these links. Both of these functions need specific CORE Uids to connect and disconnect. The last function allows the user to send text messages across CORE to all links connected to the application. At startup, this application sends a rule to connect a link to itself, so if nothing else, this application "echos" messages to itself.

This application took approximately one hour to write. The bulk of the time was spent on how to develop a good console (Xterm or Command Prompt) user interface for the user. The code shows basic examples of how to create rules and links. This application also shows how to instantiate the Agent class and send messages and receive messages with very few lines of code.

B.2.2 CommandWindowDisplayApp

This application is a follow-up to the previous program. This only acts as display console for other applications to link and send data to it. For example, we can start up this application, start up the CommandWindowEchoApp, and then we finally link the echo application to the display application. After linking, anything typed and sent from the CommandWindowEchoApp shows up on this display application.

As an extension and practical application to this application, one can imaging this display console to be a monitor or projector. Some service or application needing video output can send raw video images over CORE to this console.

Currently, we are also working on a VNC CORE agent to allow multiplexing of images from multiple sources (VNC servers) to one display (VNC client).

B.2.3 WinampAgent

Winamp is a music player created by Nullsoft. Our application here allows a user to control a Winamp player that is connected to CORE using this agent. Once connected, a user can control Winamp by sending String messages through CORE, such as "play," "stop," and "next." This can be done using the CommandWindowEchoApp or the SpeechBuilderWinampAgent below.

Once again, the agent application took almost no time at all to write. The bulk of the time was spent finding and figuring out how to use the Winamp API and then writing the C/JNI code to talk to Winamp. Once done, however, it was neat to see us being able to control music on our computer from possibly anywhere in the world.

B.2.4 SpeechBuilderWinampAgent

This cool application allows us to use spoken words to control Winamp over CORE. Basically, this is the same as the CommandWindowEchoApp except that input is from voice instead of keyboard. The raw speech information goes over to a local machine and is processed by SpeechBuilder. SpeechBuilder then sends back a Frame [39] object, which contains a list of possible phrases spoken (according to some pre-programmed domain), to the application to process and send over CORE. For our demonstration, we used the application on a Compaq iPAQ 3760 running Java 1.3 and Linux Familiar v0.51. This application can run on other platforms; however, it is limited by what platform SpeechBuilder has been developed for.

Once again this application took almost no time at all to write since it only sends simple String messages across CORE. The learning curve, here, comes from learning how to use SpeechBuilder and the API to parse and understand Frame objects.

This application could be generalized to the name "SpeechBuilderAgent" since there really is nothing Winamp-specific to it. We can take almost the same java code and SpeechBuilder domain to make this control a Microsoft PowerPoint presentation with similar commands, such as play, stop, next, and previous. As discussed in section A.1.1 about the Presentation Manager, CORE allows for many possibilities.

Appendix C: General

C.1 Starting and Running CORE

```
   /**
    * constructors call this class to initialize the CORE router
    * (tested and working)
    *
    * @exception Exception
    */
   private void init()
   {
      logger.debug("Entering StartCore init code");

      // instantiate the clientThread vector
      clientThreads = new Vector();


      // then create a NodeManager
      coreNodes = new NodeManager();
      // first create a LinkManager
      coreLinks = new LinkManager(coreNodes);
      // then create a RuleManager
      coreRules = new RuleManager(coreNodes, coreLinks);

      try
         {
            // finally make the Router
            coreRouter = new Router(CORE_PORT);
            // then create the MessageRouter
            coreMessageRouter = new MessageRouter(coreRouter, coreRules,
coreNodes, coreLinks);
            // add the listener to the core router
            coreRouter.addListener(coreMessageRouter);
            // add the control from the router to the rule manager to send
messages
            coreRules.setRouterControl(coreRouter.getControl());
         } catch (IOException e)
            {
               logger.error("IOException thrown while trying to start up a
router for CORE...", e);
               System.exit(-1);
            } // end of try-catch


      logger.debug("Exiting StartCore init code");
   }
```

```
/**
 * this starts CORE
 *
 * @param
 */
public void startCore()
{
   run();
}


/**
 * this is the main run method
 * (tested and working)
 *
 */
public void run()
{
   // this next call will start the rule processing!
   coreRules.startRuleProcessing();  // this is a daemon thread

   logger.info("CORE server listening on port: " + CORE_PORT);

   // does this block on the run???
   try
   {
      // this next call will start the core router!
      coreRouter.run();
      logger.info("Router.run() exited");
   }
   catch (IOException e)
   {
      logger.error("Some IOException... exiting...", e);
      throw new RuntimeException("IOException received");
   } // end of try-catch

}
```

C.2 Starting and Connecting with Agent

```
public static void main (String[] args)
       throws UnknownHostException, IOException
   {
       if (args.length != 2)
       {
          System.err.println("This main method needs two arguments...  \nUSAGE:
java oxygen.core.agent.Agent <host> <port>");
          System.exit(-1);
       } // end of if ((args.length != 2)

       String host = args[0];
       int port = (new Integer(args[1])).intValue();
       //     Agent myAgent = new Agent("ruble.lcs.mit.edu", 25);
       Agent myAgent = new Agent(host, port);
       myAgent.setNameSpecifier(new NameSpecifier("[first = orlando [last =
leon]]"));
       myAgent.connectAndRun();

       int messageCount= 1;
       while (true)
       {
          // sample test for sending messages and receiving
          myAgent.sendString("Message_" + Integer.toString(messageCount++));
          logger.debug(new String(myAgent.receiveAndBlock()));
       } // end of while (true))
   } // end of main ()
```

C.3 BluetoothAgent.java

```
package oxygen.core.bluetooth;

import oxygen.core.agent.*;
import oxygen.core.link.*;
import oxygen.core.message.*;
import oxygen.core.node.*;
import oxygen.core.rule.*;
import oxygen.core.router.*;
import ins.namespace.NameSpecifier;

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


/**
 * <p>Copyright: Copyright (c) 2002</p>
 * @author   <i>modified by </i><a
href="mailto:owl@mit.edu">Orlando Leon</a>
 * @version 1.1 5-8-2002
 */

import com.atinav.standardedition.io.*;

// bluetooth related imports
import javax.bluetooth.*;
import javax.obex.*;


public class BluetoothAgent extends ServerRequestHandler
implements DiscoveryListener
{
       /** this variable is for Log4J logging*/
   private static final org.apache.log4j.Category logger
= oxygen.core.CoreConfig.getLogger("oxygen.core",
".//oxygen//config//log4j.config");
   /** this is the agent that all applications/services
<b><i>must</i></b> implement */
   Agent myAgent;
   /** these are the host and port variables for the
CORE server */
   private String host;
   /** these are the host and port variables for the
CORE server */
   private int port;

   // this will hold the bluetooth devices => <key =
btAddress, value = null if not connected-else ns>
   Hashtable localBtDevices;

   // current connection object to spawn thread to
   Connection currentConnectionRequest;

   // this is the StreamConnectionNotifier for the
server
   StreamConnectionNotifier serverNotifier;


    private boolean inquiryCompleted    = false;
    private boolean authenticate        = false;
    private RemoteDevice[] devices =  new
RemoteDevice[10];
    private int count = 0;
    private String connectionURL = null;
    private ServiceRecord[] records = null;

  public BluetoothAgent(String host, int port)
      throws UnknownHostException, IOException
    {
          init(host, port);
    }

   /**
    * the Bluetooth client code uses this to search for
a CORE BluetoothAgent server
    *
    */
```

```
   public String searchDevices(NameSpecifier ns) throws
Exception
   {
      // need to do something with the NameSpecifier
==> not implemented
      // yet!  STEPS: extract necessary characteristics
from the
      // NameSpecifier (easy) ... and perform a table
lookup on
      // Bluetooth devices to find a similar or
matching UUID (see
      // javax.bluetooth.UUID)

      LocalDevice ld = LocalDevice.getLocalDevice();
      DiscoveryAgent da = ld.getDiscoveryAgent();
      logger.debug("Starting device inquiry...");
      da.startInquiry(DiscoveryAgent.GIAC, this);
      // wait till the device enquiry is completed

      synchronized(this){
         this.wait();
      }

      // we may need to change this attribute set to
match what we need for CORE
      int[] attrSet = null; //{0,3,4};
      UUID[] uuids = new UUID[1];

      // this next line is where the NameSpecifier
comes in handy!!!
      uuids[0] = new UUID("1116", true); // change 1116
because that searches for a network access point
specifically
      logger.debug("\nSearching for BluetoothAgent
Service...\n");

      for(int i = 0; i < count;i++)
      {
         logger.debug("\nSearching for Service @ " +
devices[i].getBluetoothAddress());
         int transactionid = da.searchServices(attrSet,
uuids, devices[i], this);
         if(transactionid != -1)
         {
            // hand off to another thread
            synchronized(this) {
               this.wait();
            }
         }
         if(connectionURL != null)
            return connectionURL;
      }
      return null;
   }


   public synchronized void inquiryCompleted(int
discType)
   {
      logger.debug("Inquiry completed... code: " +
discType);
      this.notify();
   }


   public synchronized void
deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)
   {
      devices[count++] = btDevice;
      try
      {
         // bluetooth addresses are only 12 in length

   //localBtDevices.put(btDevice.getBluetoothAddress(),
null);
      }
      finally
      {
      }
      logger.debug("New Device discovered : "+
btDevice.getBluetoothAddress());
   }
```

```
   public synchronized void servicesDiscovered(int
transID, ServiceRecord[] servRecords)
   {
   if(servRecords.length == 0)
       {
           synchronized(this){
           this.notify();
           }

       }
       records = new ServiceRecord[servRecords.length];
       records = servRecords;
     for(int i=0;i<servRecords.length;i++){
       connectionURL =
servRecords[i].getConnectionURL(1,true);
       logger.debug("Connection url :"+connectionURL);
           if(connectionURL != null){
               // put it in the hashtable
               try
               {
                   // we can use
                   //
javax.microedition.io.Connector.open(connectionURL) to
                   // open a connection //In the case of a
Serial Port service
                   // record, this string might look like
                   //
"bt_spp://0050CD00321B:3;authenticate=true;encrypt=false
;master=true",
                   // where "0050CD00321B" is the
Bluetooth address of the
                   // device that provided this
ServiceRecord, "3" is the
                   // RFCOMM server channel mentioned in
this ServiceRecord,
                   // and there are three optional
parameters related to
                   // security and master/slave roles.
                   //localBtDevices.put(connectionURL,
null);
               }
               finally
               {
               }
               synchronized(this){
                   this.notify();
               }
               break;
           }
       }
   }


   public synchronized void serviceSearchCompleted(int
transID, int respCode)
   {

   if(respCode==DiscoveryListener.SERVICE_SEARCH_ERROR){

   System.out.println("\nSERVICE_SEARCH_ERROR\n");
    }

if(respCode==DiscoveryListener.SERVICE_SEARCH_COMPLETED)
{

   System.out.println("\nSERVICE_SEARCH_COMPLETED\n");

    }

if(respCode==DiscoveryListener.SERVICE_SEARCH_TERMINATED
){
          System.out.println("\n
SERVICE_SEARCH_TERMINATED\n");
    }
    if(respCode ==
DiscoveryListener.SERVICE_SEARCH_NO_RECORDS){
          System.out.println("\n
SERVICE_SEARCH_NO_RECORDS\n");
       synchronized(this){
          this.notify();
       }
          System.out.println("\n
SERVICE_SEARCH_NO_RECORDS\n");
    }
```

```
      if(respCode ==
DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE)
          System.out.println("\n
SERVICE_SEARCH_DEVICE_NOT_REACHABLE\n");


   }



//////////////////////////
//////////////////////
//////////////////////////
//////////////////////
//////////////////////////
/////   CORE start code
//////////////////////////
//////////////////////
//////////////////////////
//////////////////
//////////////////////////

   /**
    * constructors call this class to initialize the
CORE router
    * (tested and working)
    *
    * @exception Exception
    */
   private void init(String host, int port)
      throws UnknownHostException, IOException
   {
      logger.debug("Entering BluetoothAgent init
code");
      // set up the agent to connect to CORE
      this.myAgent = new Agent(this.host, this.port);
      this.myAgent.setNameSpecifier(new
NameSpecifier("[Device=BluetoothAgent][core=true[uid="+
myAgent.getUid()+"]]"));
      this.myAgent.connectAndRun();
   }
```

```
   /**
    * this internal class will spawn off new threads
Agent threads to handle Bluetooth connections
    *
    */
   private class BluetoothThread extends Thread
   {
      /** this is the agent that all
applications/services <b><i>must</i></b> implement */
      Agent btAgent;

      /** this is the input stream from the bluetooth
client */
      DataInputStream is;

      /** this is the output stream from the bluetooth
client */
      DataOutputStream os;

      /** this is the message to send to CORE */
      StringBuffer toCore = new StringBuffer();
      Message coreMessage = null;

      /** this is CORE's response */
      byte[] coreResponse;


      // for devices searching for CORE
      public BluetoothThread(StreamConnection request)
         throws UnknownHostException, IOException
      {
         logger.debug("Spawning off new
BluetoothThread client");

         is = new
DataInputStream(request.openInputStream());
         os = new
DataOutputStream(request.openOutputStream());


         btAgent = new Agent(host, port);
```

```
          // NOTE: need to figure out how to set what
type of device this BT device is after connection!
          StringBuffer deviceType = new StringBuffer();
          // PUT: put code here to read the device type
from the BluetoothClient (see the run(String, String)
method)
          // check with is.available() ... and do a
deviceType.append(is.read()) until we have a complete
message

          this.btAgent.setNameSpecifier(new
NameSpecifier("[Device=" + deviceType.toString() + "
[Type=Bluetooth]][core=true[uid="+
btAgent.getUid()+"]]"));
          this.btAgent.connectAndRun();
       }

       // for devices CORE queries for
       public BluetoothThread(String url, NameSpecifier
ns)
          throws UnknownHostException, IOException
       {
          logger.debug("Spawning off new
BluetoothThread client");
          btAgent = new Agent(host, port);
          // need to start connection to Bluetooth
object
          StreamConnection c =
(StreamConnection)Connector.open(url);
          is = c.openDataInputStream();
          os = c.openDataOutputStream();
          this.btAgent.setNameSpecifier(new
NameSpecifier(ns.toString() +
"[Type=Bluetooth][core=true[uid="+
btAgent.getUid()+"]]"));
          this.btAgent.connectAndRun();
       }


       public void run()
       {
          try
          {
             while(true)
             {
                // do some processing here... listening
for stuff
                if (is.available() != 0)
                {
                   toCore.append((char)is.read());
                }

                // if we have a complete message...
send it to CORE!
                // *** put an if statement here to
check the BT device class and read comment below
                coreMessage =
Message.checkString(toCore);
                if (coreMessage != null)
                {
                   // may need to do a simple device
check here... if it's
                   // something like a BT microphone...
it cannot package its
                   // data in Message objects...  so it
must just be a raw
                   // message to CORE!  use
btAgent.sendRaw(byte[]) in this
                   // case... just drop the
Message.checkString and send the whole StringBuffer as
bytes
                   btAgent.sendMessage(coreMessage);
                   coreMessage = null;
                }

                // now try to listen for CORE responses
in this loop as well...
                coreResponse = btAgent.receive();
                if (coreResponse != null)
                {
                   // we may need to check this message
and check the
                   // device...  for instance some
devices (i.e. headsets)
```

```
                    // may only be able to receive raw
bytes...  use
                    // os.write(byte[] b, int off, int
len) in this case
                    // ** Will have to extend Agent to
change the receive() method to return any bytes on the
array
                    os.write(coreResponse);
                    os.flush();
                    // no need to reinitialize
coreRepsponse...
                }

                // sleep to save processor time
                sleep(50);
            }
         }
         catch(InterruptedException e)
         {
            logger.error("Error while sleeping..");
         }
         catch(IOException e2)
         {
            logger.error("Exception while
reading/writing data with Bluetooth client");
         }
         finally
         {
            try
            {
               is.close();
               os.close();
            }
            catch(IOException e1)
            {
               logger.error("Error while closing
connection to Bluetooth device...");
            }
         }
      }

   }
```

```
   /**
    * this internal class will listen to CORE for
service discovery queries and spawn off new threads
    *
    */
   private class BluetoothDiscoveryListener extends
Thread
   {
      public BluetoothDiscoveryListener()
      {
         logger.debug("Starting DiscoveryListener
Module for Bluetooth");
      }

      public void run()
      {
         NameSpecifier coreQuery;
         String coreResponse = new String();

         while(true)
         {
            // sit and wait until we hear from CORE
            // though this is not
            // implemented yet... this next line says
that CORE sends a
            // NameSpecifier DIRECTLY to this Agent's
buffer...  we can
            // change this message format if desiredd
            coreQuery = new NameSpecifier(new
String(myAgent.receiveAndBlock()));
            // process the NameSpecifier and see if we
have a device
            if (coreQuery != null)
            {
               try
               {
                  String connectionURL =
searchDevices(coreQuery);
               }
               catch(Exception e)
```

```
                {
                   logger.error("I have to catch one of
these lame plain exceptions", e);
                }
                   if (connectionURL != null)
                {
                   coreResponse = "YES";
                   // connect device to CORE
                   try
                   {
                      new
BluetoothThread(connectionURL, coreQuery);
                   }
                   catch(IOException e1)
                   {
                      logger.error("Unknown CORE
host... maybe CORE died?");
                   }
                }
          else
                {
                   coreResponse="NO";
                }
             }

             // send the response to CORE
             myAgent.sendResponse(coreResponse);

             try
             {
                sleep(100);
             }
             catch(InterruptedException e)
             {
                logger.warn("Thread interrupted while
sleeping");
             }
          }
       }

   }
```

```
   /**
    * this is the BluetoothAgent run method that will be
the listener/service discovery side of the class
    *
    */
   public void run()
   {
      // start up a new thread to run the discovery
listener
       (new BluetoothDiscoveryListener()).run();

       // this method should actually only run as long
as CORE is running  =)
       for(;;)
       {
          StreamConnection c = null;
          // check to see if there are any CORE requests
for server information
          ///
          logger.warn("Service discovery not implemented
yet from CORE to BluetoothAgent");
          ///

          try
          {   //
btgoep://[localhost]:UUID[;][name=<service name>]
             // UUID=1116 represents a "network access
point"
             serverNotifier =
(StreamConnectionNotifier)Connector.open("bt_spp://local
host:1116;name=CORE");
             //SessionNotifier sn =
(SessionNotifier)Connector.open("btgoep://localhost:1116
;name=CORE");
             logger.debug("Waiting to process client
requests...");
             // this returns a Connection object... see
com.atinav.standardedition.io.Connection OR
javax.microedition.io.Connection
             // grab this and process...  put this in a
thread and in a while loop!
```

```
            c = serverNotifier.acceptAndOpen();
            LocalDevice device =
LocalDevice.getLocalDevice();
            ServiceRecord rec =
device.getRecord(serverNotifier);
            // this integer has to be changed to match
the service CORE is providing
            rec.setDeviceServiceClasses(0x1116);
            device.updateRecord(rec);

            logger.debug("Bluetooth client attempting
to connect!");
            // it wants to connect to CORE, do it!
            try
            {
               new BluetoothThread(c);
            }
            catch(IOException e1)
            {
               logger.error("Some IO Exeption");
            }


            //currentConnectionRequest =
sn.acceptAndOpen(this);
            //RemoteDevice btDevice =
RemoteDevice.getRemoteDevice(currentConnectionRequest);
            // commenting out the storage of this
client address...  only store slave devices

   //localBtDevices.put(btDevice.getBluetoothAddress(),
currentConnectionRequest);
            // process this connection ==>
this.onConnect(...) is called
         }
         catch (IOException ex)
         {
            logger.debug("ERROR: Failed opening
connection");
            return;
         }
      } // end of while loop... go back in from the
beginning of the loop
   }



//////////////////////////
//////////////////////
//////////////////////////
//////////////////////
//////////////////////////
/////   MAIN code
//////////////////////////
//////////////////////
//////////////////////////
//////////////////
//////////////////////////

   public static void main(String[] args) throws
Exception
   {
      logger.debug("Entering main code...");
      if (args.length != 2)
      {
         logger.error("USAGE: java
oxygen.core.BluetoothAgent <CORE-host-address> <CORE-
port>");
         System.exit(-1);
      }

      String host = args[0];
      int port = (new Integer(args[1])).intValue();

      BluetoothAgent m=new BluetoothAgent(host, port);

//     // setting the Port setting using BCC
//     BCC.setPortName("COM1");
//     BCC.setBaudRate(57600);
//     BCC.setConnectable(true);
//     BCC.setDiscoverable(DiscoveryAgent.GIAC);
//     // this tests to see if we can get the local
bluetooth device
```

```
//     LocalDevice.getLocalDevice();

       m.run();
   }


}
```

C.4 BluetoothClient.java

```
package oxygen.core.bluetooth;

import oxygen.core.link.*;
import oxygen.core.message.*;
import oxygen.core.node.*;
import oxygen.core.rule.*;
import oxygen.core.JsxHelper;
import ins.namespace.NameSpecifier;

import java.lang.*;
import java.io.*;
import java.util.*;
import com.atinav.standardedition.io.*;
import javax.bluetooth.*;

/**
 * This class shows a simple client application that
performs device
 * and service
 * discovery and communicates with a BluetoothAgent
server to show how the Java
 * API for Bluetooth wireless technology works.
 */
public class BluetoothClient implements
DiscoveryListener {

       /** this variable is for Log4J logging*/
   private static final org.apache.log4j.Category logger
= oxygen.core.CoreConfig.getLogger("oxygen.core",
".//oxygen//config//log4j.config");

   private boolean inquiryCompleted    = false;
   private boolean authenticate        = false;
   private RemoteDevice[] devices =  new
RemoteDevice[10];
   private int count = 0;
   private String connectionURL = null;
   private ServiceRecord[] records = null;


    /**
     * Creates a BluetoothClient object
     *
     */
   public BluetoothClient()
      {
      }


   /**
    * the Bluetooth client code uses this to search for
a CORE BluetoothAgent server
    *
    * @return this method returns the URL of a CORE
BluetoothAgent server if one exists
    */
   public String searchDevices() throws
BluetoothStateException, InterruptedException
   {
      /*
       * Retrieve the local Bluetooth device object.
       */
      LocalDevice ld = LocalDevice.getLocalDevice();
      /*
       * Retrieve the DiscoveryAgent object that allows
us to perform device
       * and service discovery.
       */
      DiscoveryAgent da = ld.getDiscoveryAgent();
      logger.debug("Starting device inquiry...");
      da.startInquiry(DiscoveryAgent.GIAC, this);
      // wait till the device enquiry is completed

      synchronized(this){
         this.wait();
      }

      // we may need to change this attribute set to
match what we need for CORE
```

```
       int[] attrSet = null; //{0,3,4};
       UUID[] uuids = new UUID[1];
       // this searches for a CORE!
       // NOTE: need to make this more specific to find
ONLY CORE...
       // SEE: javax.bluetooth.UUID and
javax.bluetooth.DeviceClass
       uuids[0] = new UUID("1116", true);
       logger.debug("\nSearching for BluetoothAgent
Service...\n");

       for(int i = 0; i < count;i++)
       {
          logger.debug("\nSearching for Service @ " +
devices[i].getBluetoothAddress());
          int transactionid = da.searchServices(attrSet,
uuids, devices[i], this);
          if(transactionid != -1)
          {
             // hand off to another thread
             synchronized(this) {
                this.wait();
             }
          }
          if(connectionURL != null)
             return connectionURL;
       }
       return null;
    }


   public synchronized void inquiryCompleted(int
discType)
    {
       logger.debug("Inquiry completed... code: " +
discType);
       this.notify();
    }


   public synchronized void
deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)
    {
       devices[count++] = btDevice;

       logger.debug("New Device discovered : "+
btDevice.getBluetoothAddress());
    }


   public synchronized void servicesDiscovered(int
transID, ServiceRecord[] servRecords)
    {
    if(servRecords.length == 0)
       {
          synchronized(this){
          this.notify();
          }

       }
       records = new ServiceRecord[servRecords.length];
       records = servRecords;
      for(int i=0;i<servRecords.length;i++){
       connectionURL =
servRecords[i].getConnectionURL(1,true);
       logger.debug("Connection url :"+connectionURL);
          if(connectionURL != null){
             try
             {
                // we can use
                //
javax.microedition.io.Connector.open(connectionURL) to
                // open a connection //In the case of a
Serial Port service
                // record, this string might look like
                //
"bt_spp://0050CD00321B:3;authenticate=true;encrypt=false
;master=true",
                // where "0050CD00321B" is the
Bluetooth address of the
                // device that provided this
ServiceRecord, "3" is the
                // RFCOMM server channel mentioned in
this ServiceRecord,
```

```
                // and there are three optional
parameters related to
                // security and master/slave roles.
                logger.debug("connectionURL of
discovered device: " + connectionURL);
             }
             finally
             {
             }
             synchronized(this){
                this.notify();
             }
             break;
          }
       }
    }


   public synchronized void serviceSearchCompleted(int
transID, int respCode)
    {

    if(respCode==DiscoveryListener.SERVICE_SEARCH_ERROR){

    System.out.println("\nSERVICE_SEARCH_ERROR\n");
     }

if(respCode==DiscoveryListener.SERVICE_SEARCH_COMPLETED)
{

    System.out.println("\nSERVICE_SEARCH_COMPLETED\n");

     }

if(respCode==DiscoveryListener.SERVICE_SEARCH_TERMINATED
){
          System.out.println("\n
SERVICE_SEARCH_TERMINATED\n");
     }
     if(respCode ==
DiscoveryListener.SERVICE_SEARCH_NO_RECORDS){
          System.out.println("\n
SERVICE_SEARCH_NO_RECORDS\n");
       synchronized(this){
          this.notify();
       }
          System.out.println("\n
SERVICE_SEARCH_NO_RECORDS\n");
     }
       if(respCode ==
DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE)
          System.out.println("\n
SERVICE_SEARCH_DEVICE_NOT_REACHABLE\n");
    }


    /**
     * this is the method that is called after CORE is
found!
     * Sends the data to the CORE server.  This method
will establish a
     * connection to the server and send the String in
bytes to the core.
     * THIS method is INCOMPLETE...  need to figure out a
better way to send and receive
     *
     *@param deviceType this is the type of device or
service this local Bluetooth device is
     */
    public void run(String coreURL, String deviceType)
       throws IOException
       {
          logger.debug("Entering run method ...");

          InputStream is = null;
          OutputStream os = null;
          StreamConnection con = null;

          try {
             /*
              * Open the connection to the server
              */
```

```
            con
=(StreamConnection)Connector.open(coreURL);

            os = con.openOutputStream();
            is = con.openInputStream();

            /*
             * Sends NameSpecifier info to CORE
             */
            // send CORE data here!
            logger.debug("Sending deviceType info to
CORE");
            os.write(deviceType.getBytes());

            // this is the for loop that we do
everything... in essence it's a while(true) loop
            // get data from BT device to send here...
whether using System.in and System.out
            // receive data from CORE here as well
            for(;;)
            {
                // put more code here
                // this should be a complete message
and os.write(...) should be a serialized version of a
message object
                // CORE needs a more formalized
serialization/deserialization technique here...
                // for now, you can use
oxygen.core.JSXHelper to serialize and deserialize until
it is deprecated and replaced

                // for is.read() need to check if a
message is complete... i.e. maybe Message.checkMessage()
            }

         } catch (IOException e2) {
            logger.error("Failed to send data");
            logger.error("IOException: " +
e2.getMessage());
         }
         finally
         {
            /*
             * Close all resources
             */
            os.close();
            is.close();
            con.close();
         }
      }



   /**
    * This is the main method of this application.  It
will send out
    * the message provided to the first core that it
finds.
    *
    */
   public static void main(String[] args) {
      BluetoothClient client = null;

      /*
       * Validate the proper number of arguments exist
when starting this
       * application.
       */
      if ((args.length != 1)) {
         System.out.println("usage: java
BluetoothClient <device-type>");
         return;
      }

      try
      {
         /*
          * Create a new BluetoothClient object.
          */
         client = new BluetoothClient();

         String coreURL = null;
         while(coreURL == null)
```

```
            {
                coreURL = client.searchDevices();
            }
            logger.info("CORE url found! url: " +
coreURL);
            client.run(coreURL, args[0]);
        }
        catch (BluetoothStateException e)
        {
            logger.error("Bluetooth device may not be
running or present...");
        }
        catch (IOException e1)
        {
            logger.error("Something happened while opening
or closing... or communicating with CORE...", e1);
        }
        catch(InterruptedException e2)
        {
            logger.error("Some threading bug happened...",
e2);
```