

Design and Implementation of a Generalized Device Interconnect

by

Ang-Chih Kao

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

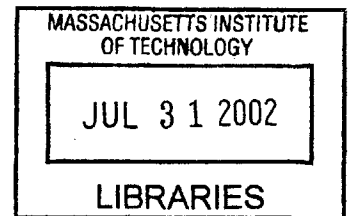
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

[Done 2002]

© Ang-Chih Kao, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis and to **BARKER**
grant others the right to do so.



Author
Department of Electrical Engineering and Computer Science

May 24, 2002

Certified by.....

Larry Rudolph

Principle Research Scientist

Thesis Supervisor

Accepted by.....

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

Design and Implementation of a Generalized Device Interconnect

by

Ang-Chih Kao

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Pervasive computing devices that communicate with each other are changing the way services are provided and utilized over a network. This thesis explores a new paradigm in establishing network connections between devices, where these devices are no longer divided into server and client, but are instead peers of each other. In this new paradigm, the flow of data is determined by third party agents, rather than by the communicating nodes. This paper describes the implementation of this network design, dubbed the Communication Oriented Routing Environment (CORE), as well as three applications of CORE that demonstrate its strengths and limitations. In summary, though successful, results with CORE indicate that the peer connection paradigm will require a reworking of current applications and design strategies in order to accommodate requirements such as authentication and client adapted data.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist

Acknowledgments

Thanks to the Amay, Shalini, Josh, Jorge, Leon, Kan, and the others in the ORG for providing a great atmosphere to work and laugh with.

Thanks to Todd Amicon for fielding all the complaints and providing all the faulty stock tips. Go Kmart!

Thanks to Cynthia Mae Chow for the encouragement and support, but most of all, her eloquent editing skills.

Thanks to Orlando Leon for being the great co-conspirator in creating CORE and making it what it is.

Thanks to Professor Larry Rudolph for the ideas and backing that made CORE, and for being an excellent advisor.

Thanks to my parents for pushing me on – from my undergraduate life, through my Master’s thesis.

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.

Contents

1	Introduction	7
2	Motivation and Background	11
2.1	Related Projects	13
2.1.1	Intentional Naming System	13
2.1.2	JINI and RMI	15
2.1.3	Metagluе	17
3	A Generalized Device Interconnect	19
3.1	Design of the CORE Network System	23
3.1.1	Composability and Automation	25
4	Applications	27
4.1	Voice-Controlled Winamp	27
4.2	VNC	28
4.3	RMI	31
5	Analysis and Conclusion	39
5.1	Analysis of Porting VNC	39
5.1.1	Access Control	39
5.1.2	Automatic Translation	40
5.2	Analysis of Porting RMI	41
5.2.1	Output Response Stream	42
5.3	Conclusion	43

List of Figures

2-1	INS in action	14
2-2	RMI in action	15
3-1	Home Entertainment Center	20
3-2	Home Entertainment Center using Metaglue	21
3-3	Home Entertainment Center using CORE-NS	24
4-1	VNC over CORE	30
4-2	RMI over CORE	33
4-3	RMI Line Wire Specification	37

Chapter 1

Introduction

This thesis is about exploring the advantages and implications of delegating control over connections from the communicating nodes to third party agents. The need for this arises from analyzing the dynamics and requirements pervasive computing device networks. We implemented the Connection Oriented Routing Environment (CORE) along with three test applications to evaluate the generalized device interconnect paradigm.

The last decade witnessed the explosive growth of computer networks, the most notable of which is the Internet. Coupled with this radically new communication medium is increased power, complexity, and most importantly, numbers of devices that were able to communicate over networks. Unlike their predecessors these network enabled devices replaced, these newer devices provide services through the network, utilize services provided by other device, or both, e.g. 802.11 or Bluetooth PDA's and cellphones can form ad hoc Personal Area Networks(PAN). By communicating with each other, these devices offer many new features without changing their own hardware or software: PDA's can browse the Internet by connecting to an ISP through the cell phone, cell phones can synchronize address books and appointments with the PDA, the PDA can automatically dial numbers on the cell phone, et cetera [7]. It is reasonable to predict that in the future, more and more devices will become "smart" (e.g. integrated with some computational power and the ability to network) including

the traditionally “dumb” devices, such as household appliances and furniture.

These networks are drastically different in both composition and operation from existing network of PC's and servers. Two of the major differences between a network of pervasive devices and a network of PC's are the number of nodes and their usage. The number of nodes in a pervasive device network is much greater. Instead of each individual having access to just one PC, which is connected to the network, a person could have multiple devices, including a PC, a PDA, a cellphone, a Bluetooth enabled pen[2], etc. Additionally, in the pervasive device network, there is no clear division into client and server. In the PC network case, dedicated servers are responsible for providing reliable access to services such as NFS or HTTP, and the client PC's are purely consumers of these services. With pervasive devices, however each of these devices can be consumers and providers of services. In the PDA and cell phone example, the cell phone provides the ability to connect to the Internet which the PDA could utilize for checking email, while the PDA provides the ability for the cell phone to synchronize contacts and appointments. In short, the power of these new network enabled devices come from their ability to either provide a service to another device or utilize services provided by other devices.

This means that the operation of these networks are much more dynamic than the traditional PC/server scenario. In the traditional PC/server case, the client is either hard-coded or manually configured to talk to a specific server. Differences in network dynamics make this impossible for networks of pervasive computing devices. First, the sheer number of connections makes hard-coding or manual configuration daunting and impractical. Moreover, devices providing services are not nearly as reliable as traditional servers, in the sense that they may leave at any time, and without warning, as a result of normal activity. The service can be resumed by a different device. This means that the server for a service can change. Additionally, completely new devices may join a network, but provide known services. These last two characteristics mean that there are no longer such things as stable, known servers, but only known services, provided by unstable peers.

A great deal of work has gone into designing a device interconnect that deals

with these problems encountered by pervasive computing devices. Some of them are examined in Section 2.1. These projects approach this problem in dissimilar ways, but they all overlook changing the method in which network connections are made, and services are utilized. Namely, these projects still hold on to idea of client/server and simply bend the paradigm around the needs of pervasive computing.

This thesis explores a new way of viewing these nodes as neither clients nor servers, but as peers creating a whole new way of viewing networks and network services. In order to do this with a myriad of different devices the control over connections needs to be moved from away from both end-points and into new third-party agents that determine the flow of data. Thus, the goal of this project is to create a novel device interconnect that delegates connection control (which client talks to which service provider¹) to third party agents, much in the way of a resource manager. Additionally, this system must be generalized to accomodate the wide variety of pervasive computing devices. An illustrative application for this interconnect is the home entertainment center, where the appliances are the network nodes, the interconnecting wires are the links, and the people who wire those appliances together, and configure them are the third party agents.

As a proof of concept, we built a Java network API that implements this new paradigm as another layer on top of TCP/IP, dubbed CORE. To find out the real-world benefits and limitations of this new concept, we created a set of three of applications that utilize CORE. The first one is a voice-controlled Winamp system. This system is composed of two agents; the Winamp agent, that allows Winamp to be controlled remotely and a voice recognition agent that utilizes the Galaxy system[23] and SpeechBuilder[24]. These CORE agents create a new system allowing Winamp to be controlled by voice. The other two applications are adaptations and modifications of two existing popular applications – Java RMI and AT&T VNC – to utilize this new paradigm.

Before the design and implementation of CORE and its applications are presente,

¹The term “service provider” is used rather than the term “server”, to distinguish the service provision as being separate from the rest of the denotations and connotations associated with “server”.

some background and previous work will be discussed to show the need for CORE. Section 2 provides background on the issues that this new interconnect solves, other projects that try to solve the same problem, and why we came up with this different paradigm. Section 3 describes the vision for this new paradigm in real world. The details of the implementation and the interfaces of CORE are presented in Orlando Leon's Master's thesis[12]. Section 4 details the design and problems encountered in creating the three demonstration applications. Section 5 analyses the strengths and limitations of this paradigm, as well as presents possible avenues for future exploration.

Chapter 2

Motivation and Background

The main difference between a network of pervasive computing devices and a network of PCs lies in the increased number of devices that provide a much broader range of services, with many more clients; more of the devices are providers of services rather than just clients, and these services are often non-reliable in the sense that the service may disappear at any time. These differences are not necessarily problems in and of themselves, but they require that applications be rewritten to specially support these differences. Just as Transmission Control Protocol (TCP) was written to isolate applications from worrying about reliable data transmission, as well as to improve interoperability, so there should be a uniform solution to help in coping with the nature of pervasive device networks.

There are generally three methods to accommodate for these differences. The first way is to provide a service discovery mechanism, which allows a device to notify either the devices around it, or a central registry, about the services it provides. Clients that are looking for particular services can then either query or receive notification about these services. A service “discovery” mechanism should not be confused with a mere “lookup” mechanism. Lookup mechanisms, like DNS and LDAP, allow a client to query a system for particular information, but it is not designed for quick updates to the system with new servers or services registering and unregistering [14]. Discovery mechanisms, on the other hand, are much more dynamic, allowing services to come and go, while keeping a relatively up to date view. There are already quite

a few systems that have been designed and implemented: some of them are Service Location Protocol (SLP), Universal Plug and Play (UPnP), and Bluetooth Service Discovery Protocol. See [3, 14] for a survey of various lookup and discovery protocols.

The second and third methods are both ways to hide the routing layer with a higher abstraction. The second tactic provides another way of routing data between the client and the service provider, essentially hiding the network layer(IP) with another layer. At the outset, this may seem redundant, when a protocol like IP suffices, but it is not a too low level of abstraction for pervasive device networks. Take the case of the Domain Name System. The formalization of heirachial naming structure for the Internet came about as means of making an email's recipient host easier to look up from the destination email addresses, and therefore easier to deliver[19]. With this heirachial naming scheme, each organization or domain administers the host names that are within its control – thus eliminating the need for email address to actually containing the address of specific hosts. Yet, in the current state of things, DNS names are looked up to arrive at an IP address, which is then used for all future communication with that host. Providing a method of routing packets not based on an IP address, which is determined by network infrastructure, but by some domain name, is much more consistent. For example, if the host in question is mobile, it can change IP addresses as it moves from one administrative domain to another. However, if the DNS entries keep pointing to the proper address of the host, then packets will reach the proper destination [22]. This idea is further examined in Section 2.1.1 on INS.

Another way of hiding the routing layer in a higher level abstraction is to provide a standard way for clients to utilize a specific service. This is essentially some form of remote procedure call, whether it be Sun RPC, RMI, SOAP, or something else. Like the previous method, it creates a layer abstraction that hides the network layer from the client. The Sections 2.1.2 and 2.1.3 describe two projects that use this method.

2.1 Related Projects

The following three sections survey some of the relevant projects that address the same general problem as this project. These other projects show how the three methods described in the previous section are applied and, what their benefits are. The point is to show how some of these projects employed various ideas, since and these ideas will either influence or be employed in the design of this project's generalized device interconnect.

2.1.1 Intentional Naming System

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.” [1] The system comprises of an application overlay network of Intentional Name Resolvers (INR), whose job is to resolve 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 that is looking for a service sends another name-specifier to an INR, who replies with a list of services that satisfy the request.

There are three points of note that differentiate this project from a run of the mill service discovery protocol. First, the method for describing and querying a service is done through an attribute-value tree dubbed, *name-specifier*. Unlike other projects such as JINI [26], the INS name-specifier is very descriptive yet constrained to fixed structure.

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

The third and most unique idea in INS is the idea of integrating name / service

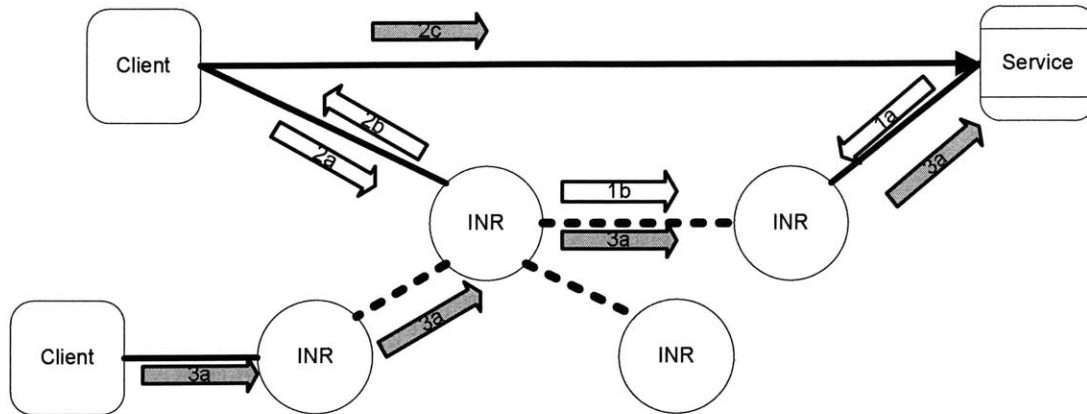


Figure 2-1: *INS in action*. The figure depicts the events that can happen in an INS setup. (1a) A service connecting to the network, locates an INR and sends its service description. (1b) The INR then propagates an update to the other INRs. (2a) A client sends a request to its INR for the service just connected. (2b) The INR responds, and (2c) the client establishes a direct connection with the service. A second client (3a) utilizes late binding by sending destined for the service to an INR, (3b) who forwards it on.

lookup with the routing of the data packets to create a level of indirection between the nodes, called *late binding*. This means that a client can simply send data with the destination designated by a name-specifier rather than an IP address. The benefit is two fold: First, the client does not have to worry about how the device is connected – this is important since the device could be a mobile client that is moving from wireless access point to access point, and changing IP address in the process. As the wireless device moves, it keeps an INR informed of the current address, such that any data packets that are destined to the mobile device is automatically routed to the new IP address. The INS system also supports two other types of routing: anycast and multicast. Anycast routes the data to any of the nodes that satisfy a given name-specifier, and multicast broadcasts the data to all of them [13]. For example, a printing service may be interested in printers that support certain capabilities, but beyond that, 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. Figure 2-1 shows an example situation of INS in action

2.1.2 JINI and RMI

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.

RMI works as follows: Upon startup, a service locates an RMI registry. The registry acts as repository of known services that clients can query. 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 .class 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, which is specified by the service. 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

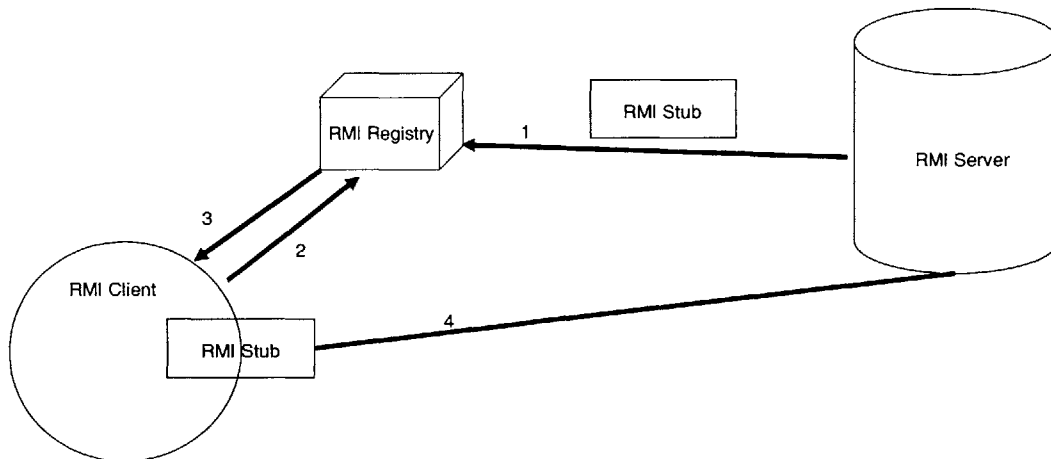


Figure 2-2: *RMI in action*. When a RMI server starts up, it export the remote object to a RMI registry by sending it the stub to the remote object(1). Later on, when a client wishes to use the remote object, it requests the stub from the registry(2). It then uses the stub it receives(3), to invoke remote methods(4).

Sun created a layer on top of RMI, called JINI, to provide extra functionality that makes it possible to build agents that dynamically find each other and organize themselves into distributed applications. Some of these features include a more powerful and flexible lookup service, and explicit support for distributed events[9].

The discovery mechanism for JINI is much improved over RMI's. Instead of services being registered and looked up by a hardcoded name as in RMI, 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 holding the attributes needs to 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. See [5] for a detailed analysis of JINI's service discovery protocol. The second important feature is the idea of distributed events, which allows the services to "push" data to clients. Whereas with RMI¹, the client is responsible for pulling (polling) data from the service, JINI now allows the servers to initiate the flow of data.

JavaSpaces

One of the services built using the JINI infrastructure is the JavaSpaces[8] data sharing mechanism. JavaSpaces 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 the data into this pool, do not have any explicit knowledge of who the consumer of that data is, if any. A service which is responsible for processing that data can choose to process that data at any time, whether it be for load, arbitration, or any other reason. Likewise, the service that takes the entry from a JavaSpace, does not need to know who generated the entry. In fact, by the time a service processes the entry, the client which inserted the entry could already be disconnected from the JavaSpaces.

"[JavaSpaces] are designed to work with applications that can model themselves as flows of objects through one or more servers." [27, pg. 4] Essentially, the JavaSpaces acts as a type of queue for the various services to take and insert entries. Another way

¹This is not to say that distributed events can not 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.

of viewing it is as a bulletin board, where clients that require services post requests and wait until their requests are processed by a service. This idea will be discussed in more detail in Section 3.

2.1.3 Metaglu

In a lot of ways, the Metaglu[18] system is similar to the Jini system. They both have a lookup system, and are based on Java and RMI. They also both have the aim of facilitating dynamically configurable systems. The difference is that Metaglu aims to be a substrate for smart-spaces[15] as part of an over-arching program to build an intelligent room. To that end, it has extra functionality, such as the ability to move agents around from one virtual machine to another, a network accessible persistent backing store, and several others. These features, while interesting are beyond the relevance of this project.

What is of particular interest, is Rascal[6], a resource manager built into the Metaglu system. This resource manager provides both resource mapping and arbitration, meaning that Rascals know which resources can satisfy a particular clients 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 on 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 who wishes to do a series of activities such as watch movies, check web pages, etc. Rascal would determine that satisfying the users request is significantly more impor-

tant 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 Metagluce, a client does not control to whom it is actually talking, rather it delegates that responsibility to a third party. This feature is unique, because it incorporates the concept that the client can be instructed about whom to talk to not just at the initial stage of finding a partner, but at any time. It is also important to note that Rascal is a “manager” of resources, not merely a mechanism for “discovery”, i.e. a client needs to carry through with what the manager says, and cannot decide to use another resource that the manager did not approve. The method by which this resource management is implemented involves client stubs, making this entire process transparent to both the resource provider and consumer.

Chapter 3

A Generalized Device Interconnect

Chapter 2 details some of the various projects that gave inspiration to the design of the generalized device interconnect described in this chapter. The core concept is that clients and servers should not be responsible for the management of connections – rather they delegate this to a third party. Even the very terminology of client and server is more reflective of the underlying management of connections than of their roles in providing or consuming services. Take the example of VNC (Virtual Network Computing)[33] and X: In X, the X server is the program that displays programs on screen while the X clients are the various programs which provide the images to display. In VNC, the designation is reversed, where the VNC server is the program generating the images and the VNC client is the software that actually displays the images. This simple example just demonstrates the point that the idea of server and client is really more reflective of their connection management roles than their role as part of a software system.

The rule of thumb that servers are managed by administrators, and clients reside on the end-user's machine is no longer relevant in the field of pervasive computing, where there are many devices, each of which could either provide or use services. Though this is not a comprehensive justification, it does suggest revisiting how connections are formed and abstracted to software.

The key idea behind this project's generalized device interconnect is to completely remove any involvement of a connection's endpoints in controlling the connection.

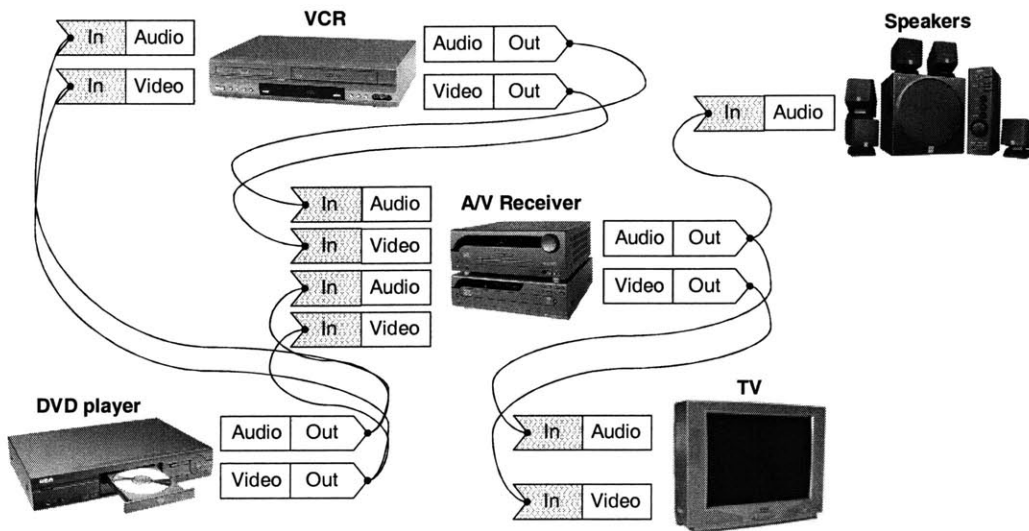


Figure 3-1: *Home Entertainment Center*. The figure depicts a home entertainment center with several devices. The connections formed by the wire determines how the system operates. In addition the A/V receiver acts as a multiplexer, that the user controls. This allows the user to grant either the VCR or DVD player control over the TV and speakers.

This goes far beyond merely performing the data routing as in INS, or redirecting connections as in Rascal – the client does not even request the services it needs. To illustrate this principle take the following example of the home entertainment center. Figure 3-1 illustrates the various components in a home entertainment center that are connected to each other. First let's explain how this would operate in Metaglugue/Rascal. These components, instead of being connected by wires, are in fact all connected to a high bandwidth network that can carry all the audio and video data amongst the devices. Furthermore, each of the devices have a software component that allows them to operate in Metaglugue. The resulting picture would look like something Figure 3-2, where each of the audio or video producing devices requests the TV or speakers, respectively, in order to accomplish their task.

Recall that in all the systems described in Section 2.1, the clients had to request services of the server. The home entertainment center example points out the oddity of this structure. Why is it the responsibility of the DVD player or VCR to request the TV and not the other way around? It makes just as much sense for a VCR to

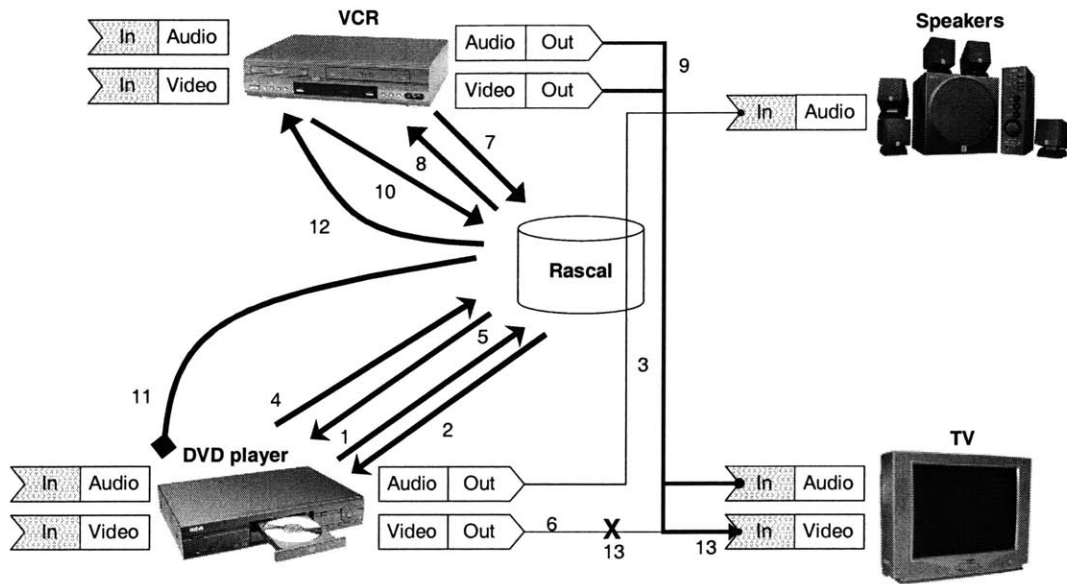


Figure 3-2: *Home Entertainment Center using Metaglué*. With Metaglué and Rascal, there are no longer wires or an A/V receiver. Instead, the resource manager, Rascal, deems which service to allocate for each client. In this drawing, the DVD player first requests speakers(1) and a video display(4). Rascal replies with the resources that DVD player should use(2,5), to which the DVD player connects(3, 6). Later on, the VCR comes along and requests the speakers(7) and a display(10). The speakers attached to the TV are free, so Rascal allocates them to the VCR(8, 9). On the other hand, there is only one display, so Rascal tells the DVD player to stop using the TV(11), and then grants the TV to the VCR(12,13).

request a display, as for the TV to request a video input. In short, there is no true sense of client in this case: both the TV and VCR provide a service depending on the point of view. So, in the model of our generalized device interconnect, the connection between the devices is arranged by a third party, much like how an owner today connects the various components together manually with wires.

The client/server paradigm has a large impact on the organization of a system, and it is short sighted to dismiss the argument against it as being only of academic interest. One major result of the client/server paradigm is in localizing of environmental and end-usage knowledge at the end-points. For example, in the INS system, each node is responsible for broadcasting an accurate description of itself, which includes its capabilities (which can be determined by the node, itself) and other details like its location or other information that is determined by some administrator. A client that

is looking for a particular device generally has to refer to these non-intrinsic properties (for example, the VCR needs to know which speaker system is actually part of the home entertainment center). In the home entertainment center, there needs to be some amount of manual configuration. Each device must be told to whom it should connect.

The Rascal system is slightly better than this, since the knowledge of which exact resource each node should get is concentrated in the resource manager. Each time new devices are added to the system (i.e. a TV), instead reconfiguring all the other devices to utilize it when appropriate (i.e. a VCR), only the resource manager needs to be configured to handle this device. This, however, raises the question as to why a client needs to explicitly request a resource when the resource manager already has information about what each client wants, and which services can satisfy a client, and the cost/benefit of each arrangement.¹

The design of the generalized device interconnect takes the concept of Rascal even further. It extends how Rascal behaves, by relieving the client from even making the initial request, delegating this to a third party as well. This is a logical step to make, given one important observation – the way resources are connected dramatically affects the purpose / use of the devices. Back to the home entertainment center in the Metaglove and INS world; when a user hits play on the DVD player, the player seeks out a display, hopefully a TV rather than a monitor, to display the image. Yet, it is also possible, that the user intended for the video and audio data to go the VCR, which is recording it to a tape. Legal issues aside, these are two possible applications that use the same set of devices. In INS, the designers for the client (the DVD player) must foresee all possible uses for the client (watch DVD and copy to VHS). Obviously, this is an undue burden on the designer of the client, and severely limits the uses of these devices. With Metaglove / Rascal, the problem is limited to reconfiguring the

¹An argument can be made that a client should request resources, before a resource manager allocates a resource for it, as the client knows best about what it needs – and these needs can change over time depending on factors such as user input, which the resource manager is not aware about. However, this is something that a resource manager, like Rascal, suffers from as well. This is also a point about the restructuring of how services are written and how resources are consumed. These points are discussed later in this thesis.

resource manager, but it is odd that a resource manager needs to be reconfigured each time the user wants to do something different.

One possible solution around this situation is to turn the VCR and DVD clients into service providers – in the sense, that they no longer request to connect a display. Instead, another agent requests the DVD and TV service, and pipes the A/V data from the DVD to the TV. For copying DVD's the agent requests the VCR service instead, and pipes data from the DVD to the VCR. This solution is much cleaner than modifying the client or resource manager to accommodate viewing versus copying a DVD, as it removes the burden of expandability from both end points and puts it into an upgradable third party agent. This is the type of solution that this generalized device interconnect promotes.

3.1 Design of the CORE Network System

The design of this generalized device interconnect (henceforth referred to as CORE-NS – Connection Oriented Routing Environment Network System – the first implementation of this new interconnect) creates a new network layer, rather than just completely abstracting away the underlying network protocols. This results in a clean interface devoid of ports or connections. The interface instead presents simple input and output streams to read and write data. To make this possible, the CORE NS is composed of two parts: the agent and CORE. CORE in turn is comprised of three parts: discovery mechanism, network infrastructure, and rule processor.

The agent simply refers to any device, service, or client that wishes to utilize CORE – true to the concept, there is no differentiation between client and server. The way these agents interact with each other bears similarities to the design of JXTA[10], with different terminology. More specifically, an agent contains a pair of I/O streams that the agent can read and write to at any time, without concern about to whom they are connected, if anyone. The output stream of one agent can be connected to the input stream of the same or another agent, with a unidirectional link. Unlike JXTA, however, which has two different pipe modes depending on how

many input streams an output stream is connected two, all links in CORE are the same – they all behave like a *Propagate Pipe* [28]. This means, that when there are two links connecting the same output stream to two input streams, each input stream receives a copy of all data that comes out of the output stream. Figure 3-3 shows an example of how agents communicate with each other.

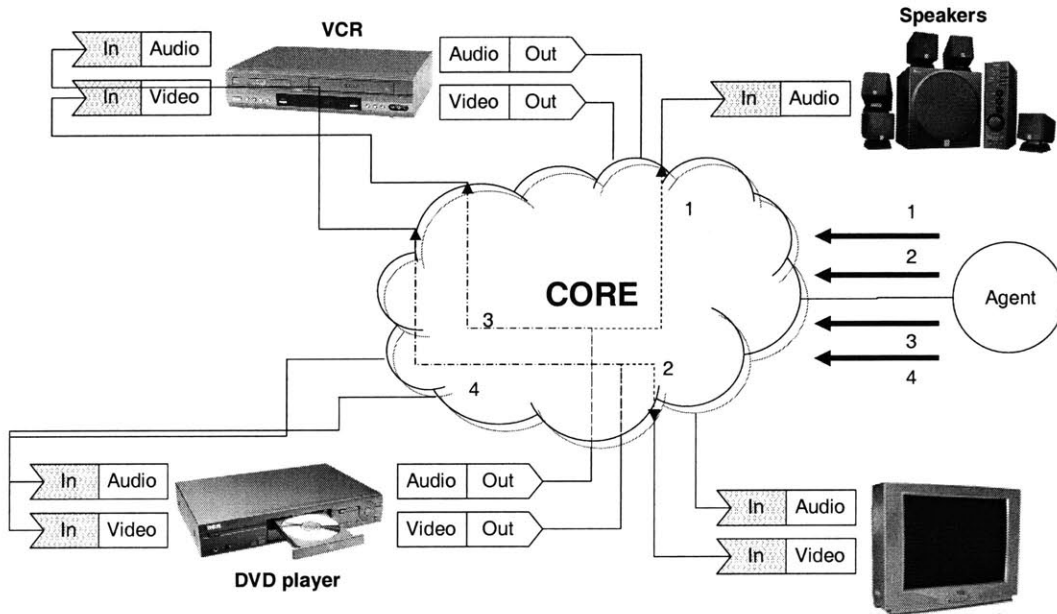


Figure 3-3: *Home Entertainment Center using CORE-NS*. With CORE-NS, all the agents and their ports (both input and output) are connected to CORE. The resource providing agents (TV, VCR, et. al) do not know who they are linked to, if any, nor do they request to be linked. Instead, a linking agent issues commands to CORE to link, the DVD audio(1) and video(2) to the speakers and TV, and later on issues the command to connect the DVD to the VCR(3,4).

The internal parts of CORE facilitate the operation of the agents as described above. In the previous figure, CORE is the large cloud that simply provides a set of functionality. For now, CORE should be viewed as this cloud, in order to focus on the design and goals of the generalized device interconnect.

The main functionality that CORE provides is an interface for third party agents to specify the arrangement of links that connect the I/O streams of other agents. Therefore, an agent, when starting up, must inform CORE about itself, including its capabilities. Agents looking to create links have access to this information via queries to CORE. It is important to note that while a TV agent can connect itself to a VCR

agent, a third agent can also connect the TV agent to VCR agent. In fact, CORE is designed to have these third party linking agents, since this is how it removes the responsibility of managing connections from the end-points. The internals of CORE consists of three parts that make this possible: the discovery mechanism, to keep track of available agents and their descriptions, the network infrastructure, to maintain the links between the streams and route data based on those links, and the rule processor, to provides the programming interface that the agents use to create and manipulate the links of other agents.

3.1.1 Composability and Automation

The rule processor provides a programming interface that is more like a scripting language rather than function calls. It is more semantically powerful than configuration templates. In the example of the home entertainment center, linking agents can link device agents in different ways to do drastically different things, without the resource agents having to do anything differently. With this in mind, it makes sense that at the heart of these third-party linking agents, should be a set of commands (in the form of rules) that CORE can understand and process.

This means that entirely new applications can be created without the need to understand the intricacies of programming agents as other systems require. Indeed, building a system with CORE is more like a game of connect the dots, than programming an application. Simply draw arrows (links) from one dot to another (agents) to create custom applications that are completely new and adapted to the users needs.

Orlando Leon's thesis[12] elaborates on the rules, their semantics, and how they are processed. The important thing is to keep in mind is that projects involving home automation, or user customized behavior, generally relies on some sort of learning, trying to predict users desires. However, if a system has an understandable representation of the state of the system, and provide simple constructs to modify the system based on events, than automation can be created and custom tailored by the average non-technical user. Though this idea is not thoroughly investigated in this thesis, it is an important benefit of this peer connection design and it is being studied by others

this research group – the Oxygen Research Group (ORG)– to make computers as pervasive and easy to use as oxygen.

Chapter 4

Applications

In order to evaluate the effectiveness of this new peer paradigm, we built three sample applications that communicate over CORE. The first application is a voice-controlled Winamp, which demonstrates the ease of building an agent. The second and third application are adaptations of two popular software packages, Java RMI and AT&T VNC, to utilize CORE.

4.1 Voice-Controlled Winamp

The voice-controlled system consists of a pair of agents – one to control Winamp, and another to transcribe speech into its text via the SpeechBuilder system[24]. This pair was the first set of agents that we built on CORE, and it mostly serves as a test, proving that the CORE NS works.

Winamp¹ is a popular MP3 player produced by Nullsoft². Winamp has built-in support for third party plug-ins and applications to control the functioning of Winamp. Building the Winamp agent mostly consisted of looking up how to control the MP3 player via its published API[34]. Most of the time spent on this agent was for writing the C code that made the appropriate method calls to the Winamp API³, and wrapping it all with a JNI interface. The actual agent code, however, only

¹<http://www.winamp.com/>

²<http://www.nullsoft.com>

³Most of the API is sending specific Windows messages to the Winamp window.

took a few minutes to create, and the entire source code is listed in the Appendix A. Decoupling the service agents from the responsibility of maintaining connections greatly shortened the development time of this agent, and alleviated the need to make an arbitrary decision as to whether this agent would be the client or the server of the voice-controlled Winamp system.

The second part of this system is an extension to a regular SpeechBuilder client. This client uses a microphone to record a user's voice command, which it sends across the network to a sever⁴. The server transcribes the voice, and returns the most likely result to the client. This client then simply retransmits this text over its output stream to CORE. Thus, if there is a link between the SpeechBuilder agent and the Winamp agent, the Winamp becomes voice controlled over the network.

In one day, we were able to interface the two disjoint systems together, solely with connections through CORE. These agents show how easy it is to create resource agents that provide resources that were not designed for use with CORE – in fact, neither of the original applications were written with the intent of distributed control and use. This is indicative of the great possibilities that this system holds for future of pervasive computers – imagine a room where all the devices (lights, fans, stereo, et al) are all automated through connections to CORE.

4.2 VNC

Virtual Networking Computing (VNC)[33] is popular system of viewers and servers that when combined, provide the same type of functionality as X – namely the ability to have the GUI of an application appear on a different machine than the one the application is running on. The most common use of VNC, though, is to remotely control entire desktops, like the Microsoft Remote Desktop Protocol. However, VNC applications can also be written to use the VNC Remote FrameBuffer (RFB)[20] protocol for its GUI, much like X.

⁴This part of the communication is done over SpeechBuilder's own network code, not through CORE

Porting either VNC or X to use CORE provides the ability to move the GUI for an application from display to display, depending on need. For example, projectors and presentation displays no longer need to be directly connected to the computer that is running the presentation software. Instead, they each run a VNC viewer, and the system running the presentation software runs a VNC server. Using CORE, the projectors can be connected to the presentation software – all through the network. There is no longer any need to worry about having the correct software drivers and cables, since all the image data is now flowing through the network.

We chose to port VNC to CORE rather than X, because of the apparent complexities of X versus VNC. The X protocol is more similar to an asynchronous RPC protocol, though not all requests require a reply[21]. VNC, on the other hand, has no sense of a request or reply, instead, the input and output streams can be treated completely independently. This means that if multiple clients are linked to the output stream of the same VNC server, there is no need for them to know about the actions of each other.

To make VNC work over CORE, there are two separate agents, the VNC Server Agent, and the VNC Client Agent. The VNC Server Agent connects to a running VNC server and to CORE, acting as the gateway between the two. The VNC Client Agent connects to CORE and waits on a port for a VNC viewer (client) to connect. This method, means that there does not need to be any modifications to the stock VNC server and viewer.

Perhaps, the final point that convinced us to work with VNC and not X, is the fact that the VNC (RFB) protocol specification is only 26 pages long, whereas the X version 11 release 6 specification is more than 6 times longer. And after all, this is a Master's of Engineering Thesis, not Ph.D.

The first problem in adapting VNC to CORE using this method is that whenever a viewer connects to a server, there is phase of connection setup that performs two things. Initially, there is a password authentication stage using challenge response. Next, the server sends a `ServerInitialisation`[20, p. 10] message that informs the viewer about the display characteristics of the desktop that server is running on – this

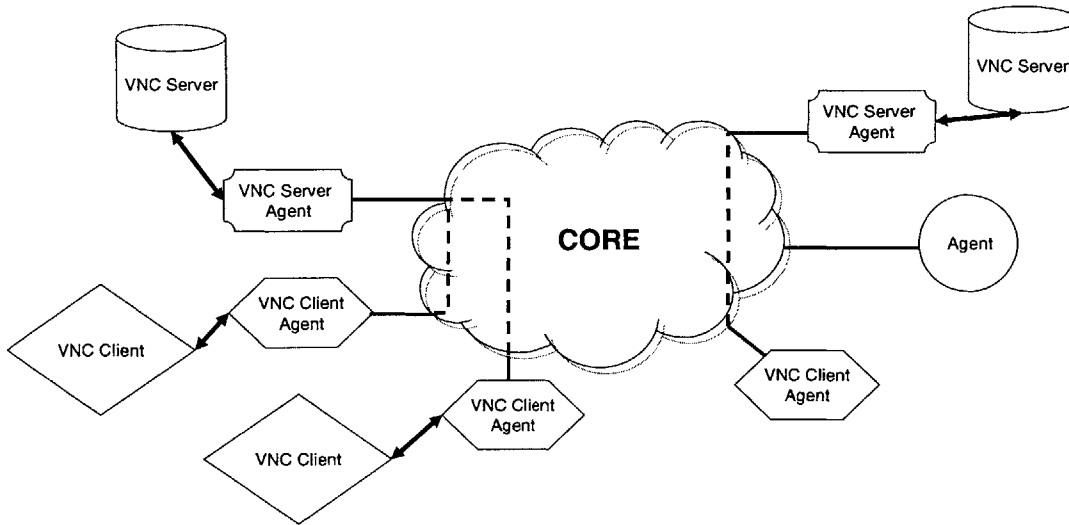


Figure 4-1: *VNC over CORE*. To port VNC to CORE, we created proxies between CORE and the VNC servers and viewers. When VNC Server Agents start up, they connect to CORE and their designated VNC server. VNC viewers connect to VNC Client Agents that are already started and connected to CORE. Linking agents link viewers to servers.

includes desktop size, bits per pixel, and an endian flag. Removing the authentication was a simple process: the VNC Server Agent behaves like a client to the VNC server, responding to the password challenge as a client. After the initial setup phase, the VNC Server Agent simply pipes all the data from its CORE input stream to the VNC server, and does exactly the reverse for the CORE output stream.

The VNC Client Agent has a similar strategy, except that here, there is a problem with the proper `ServerInitialisation` message that the agent should send to the viewer. Indeed, this is an inherent incompatibility between the design of VNC and CORE. Central to the generalized device interconnect is the idea that it does not matter which agents are linked together, and it does not matter if those links change. This is not a problem with a home entertainment center, since all the devices exchange audio and video data in the same format. This is not an available luxury in inter-device communication, when something so fundamental as little-endian versus big-endian severely affects the communication between two devices. The proper solution is that all VNC viewers should be able to accept `ServerInitialisation` messages at anytime during a session. This way, when a client switches the server that it is

linked to, it can recover by waiting for the next `ServerInitialisation` message.

The client, itself, sends a message similar to the `ServerInitialisation` called `SetPixelFormat`. This message is like the server side equivalent, though it does not contain the desktop size. In a VNC system, the data from the server is sent in the format described by the `ServerInitialisation` until the client supersedes it by sending a `SetPixelFormat` message to the server. This client message creates a different problem than the server problem: when a client sends this message, it expects that all subsequent data from the server will be sent using this format. Other clients which are also linked to the output stream of the server, suddenly receive data in the wrong format, though there is no way they can tell, other than the garbled data.

This all means that the design of VNC is inherently limited to just one client and server for each connection. Even then, there needs to be modifications to the VNC client for it to handle the equivalent of a `ServerInitialisation` message at any time in the data stream, and not just at initialization. The correct solution would be to design a new system from the ground up that is built to support multiple clients with the same pair of input and output streams.

In the end, the best results that could be achieved without modifying either the VNC server or client is to allow multiple VNC viewers to connect to the same VNC server, and allow these viewers to control the remote desktop, but not view it. For the reasons, stated above, we were unable to get the remote desktop to display properly on any of the clients. Section 5.1 discusses the lessons learned from this endeavor.

4.3 RMI

We next tackled the project of making Java's Remote Method Invocation (RMI) perform all its network communications over CORE. The goal of this was to allow the current slew of applications using RMI and RMI-based systems, such as Jini or Metaglove, to migrate over to CORE. This way, CORE reaps the benefit of a tried and true RPC system, while providing to these applications the advantages of the peer

connection paradigm. CORE offers the ability to transparently connect an agent to a similar service if the current service suddenly fails. CORE also allows service agents to be linked in completely novel ways (though still satisfying the object type safety) to create new applications.

As mentioned in Section 2.1.2, when an RMI remote object server starts up, it first contacts a RMI Registry and uploads an instance of the client stub to the RMI Registry. This part of the process, known as binding, is implemented by classes in the package `java.rmi.registry` and the separate program `rmiregistry`, both of which do not have documented ways of changing its operation – though Sun does provide the source code to do both. Also, changing these components would require that all the RMI clients and servers would need to use a modified version of the JDK - a troublesome and undesirable solution.

It is not a great problem to use the RMI Registry to start the first RMI remote object server and client talking to each other. Sun urges the use of the RMI Registry as purely a bootstrap mechanism, and that the object returned by the RMI Registry be a factory object, whose purpose is to return other RMI Remote Objects[31, 29].

After passing the initial object around, JDK 1.2 and on provide a mechanism for controlling the way data is sent back and forth between the remote object server and stub. This support is called Custom Socket Factory, where the client stub, which the server passes to the RMI Registry, contains a factory that creates sockets for the client to use to communicate with the server. Since this method is supported by the JDK, it does not require any modifications to the clients, including the RMI Registry, and the required modifications on the Remote object are minimal to none. All that needs to change is the method call to the `java.rmi.server.exportObject()`, to supply the custom RMI socket factories[32].

Wrapping the agent API into subclasses of the Java `Socket`, `ServerSocket`, `InputStream`, and `OutputStream`, was not hard at all. The result is a pair of factories that create instances of the `Agent` class from CORE, instead of TCP sockets, as the standard socket factories do. The constructor for these socket factories require the

name-specifier⁵ which the agents created by the socket factory announce to CORE. Additionally, the client socket factory constructor requires the name-specifier of the server, such that when the agent created by the client custom socket factory connects to CORE, it also links to the proper server's CORE agent.

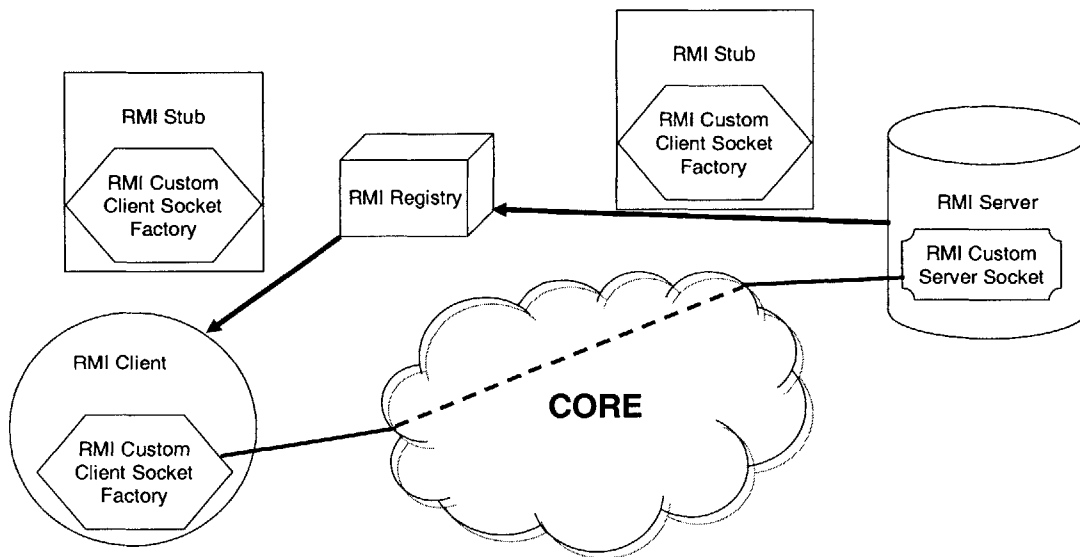


Figure 4-2: *RMI over CORE*. Getting RMI communicate using CORE is done through a pair of Custom Socket Factories. When a RMI server starts up, it connects to CORE with a custom server socket(1) that is a wrapper around the CORE agent stub. The RMI server also contacts a RMI registry to upload the client stub along with a custom client socket factory(2). Later on, a client requests the stub for the server(3). The RMI registry gives to the client the stub and socket factory(4). Finally, the client uses the custom client socket factory to create a socket that is also a wrapper around the CORE agent stub, to talk to the RMI server through CORE(5).

Alas, though all the aforementioned classes worked as they should, RMI just would not work over CORE. When a client tries to acquire a reference to the instance of the remote object, the client just freezes. Using copious amounts of debug statements, we were able to find out the state of the system was as follows: The remote object is successfully able to export itself, meaning that it does create a `ServerSocket` using the custom socket factory, and it does properly connect to CORE. The remote object also successfully transmits its stub to the RMI registry, and as expected the RMI registry has a copy of the client custom socket factory. Something unexpected

⁵The agent addressing/naming mechanism uses the name-specifier format from INS. Refer to Orlando Leon's Master's thesis for implementation details[12].

happens at this time, when the RMI registry actually creates a client socket using the factory, and connects to the remote object server through CORE. After passing some data back and forth, the remote object server reports that the binding in the RMI registry was successful, and ready for clients.

The RMI registry makes calls to the remote object server for the purpose of its distributed garbage collector which guarantees that the server object will always be available as long as any remote reference still exists[30]. Knowing, this there is still no reason why the client trying to use the remote object should simply freeze. The RMI line wire protocol[30, p. 83-98] combined with some network traces showed that the cause was the fact that the default method for communicating between an RMI client and a server is to use the multiplex protocol. In both the multiplex and stream protocols, the client and server maintain a connection for multiple method invocations and responses. The difficulty arises when the connection is stateful, because of an initial set-up phase that is not repeated for every method invocation. This means that, for each pair of server/client, there needs to be a separate connection, which runs counter to the whole goal of this new peering connection, where both endpoints are abstracted from the number and type of connections.

Unfortunately, there was no documented way to force RMI to not re-use a socket, but looking through the source code, there was a package level class that allows just that. The server and client would now create a new connection for each method invocation and response. The difficult part was to simulate the establishing and destruction of multiple connections, when there was in actuality, only one constant connection to CORE. The most similar behavior that RMI exhibited was in dealing with firewalls, by wrapping each RMI calls and response in HTTP request and requests. We examined the source code that accomplishes this, and then created the methods `writeNotify` and `readNotify` that are invoked each time data is written to or read from the socket, respectively.

Algorithm 1 shows the simplified pseudo-code for the `readNotify` method. The basic check that this pair of methods seeks to enforce is that after data (a request) is read from the socket, and data (a response) is written to the socket, that the socket

Algorithm 1 `Socket.readNotify()`

```
1: if this.nextAccept = true then
2:   throw new IOException("Socket is closed.")
3: end if
4: this.read ← true
5: if this.write = false then
6:   this.readFirst ← true
7: else
8:   if this.readFirst = true then
9:     if this.nextAccept = false then
10:    this.nextAccept ← true
11:    if this.server! = null then
12:      this.server.notify()
13:    end if
14:  end if
15:  throw new IOException("Socket is closed.")
16: end if
17: end if
```

Variables:

this.read if data has been read from this socket

this.write if data has been written to this socket

this.readFirst if data was read from this socket before data was written

this.nextAccept if a new connection has been created

this.server pointer to a `ServerSocket` if available

Methods:

this.server.notify() tells the `ServerSocket` to simulate a new connection

should not be used to read again(for another request). This ensures that a server does not confuse two consecutive requests as coming from the same client, when they are in fact coming from two different agents. Throwing an `IOException` at lines 2 and 17, informs the RMI system that this socket should be discarded. The extra clause in lines 11-13 handles the case when this `Socket` is created from a `ServerSocket`. Line 12, causes the `ServerSocket` that created this `Socket` to create a new one, for the RMI system to use.

This design makes the assumption that there is no negotiation phase to the communication – that each connection only contains a request followed by a response. Though it is unclear from the RMI line wire protocol whether this is actually the

case, these assumptions were also made by the HTTP firewall tunnel code for RMI.

Algorithm 2 `Socket.writeNotify()`

```
1: if this.nextAccept = true then
2:   throw new IOException("Socket is closed.")
3: end if
4: this.write ← true
5: if this.read = false then
6:   this.readFirst ← false
7: else
8:   if this.readFirst = false then
9:     if this.nextAccept = false then
10:      this.nextAccept ← true
11:      if this.server! = null then
12:        this.server.notify()
13:      end if
14:    end if
15:    throw new IOException("Socket is closed.")
16:  else
17:    writeTimerSet()
18:  end if
19: end if
```

Methods:

`writeTimerSet()` starts/restarts the timer to finish writing

Algorithm 2 shows the operation of `Socket.writeNotify()`, which is slightly different from `Socket.readNotify()`. It has the additional line 17, which is necessary to deal with case where the RMI server writes a response, and does not close the socket, but does not use it to read more data either. Instead, it is waiting for the `ServerSocket` to accept another connection and create another `Socket`. The method `writeTimerSet()`, creates a timer that invokes `this.sever.notify()` at timeout. The idea here is to guess when the RMI server has finished writing the response for the client, by waiting for a set amount of time (100ms in the current implementation), and see if data was written in that period. If not, then consider the socket as dead. The proper way would be for the server to actually close the socket, but since we do not want to modify the RMI implementation, it is necessary to have this timer.

The result was that there were no long standing RMI connections, and each

method invocation contained all the necessary information to process the request. This now allows an arbitrary number of clients to be use an RMI sever, even though there is actually only one pair of I/O stream with which the server talks to the clients. However, when we change the RMI server that a client is linked to, the client is no longer able to use the remote object it has a reference to. The reason for this is in the RMI line wire protocol that dictates how a method call is encoded. The reason is that the *ObjectIdentifier* shown in Figure 4-3, contains a number that is unique to each instance of the same object. Therefore, two separate and identical RMI servers exporting the same object will have different *ObjectIdentifiers*, and hence a client that is using one server cannot switch to the other server without re-requesting a remote reference.

<p><i>CallData:</i> <i>ObjectIdentifier Operation Hash Arguments_{opt}</i></p>

Figure 4-3: *RMI Line Wire Specifiaction*. The specification for the encoding of the method to invoke. The *ObjectIdentifier* indicates the object, with the method denoted by *Operation*. *Arguments* are the arguments for the invoked method.

This problem is not present for all RPC protocols, but is present in RMI because the server is stateful. For example, an NFS Server[17] over RPC[16] is stateless, so a client can switch from NFS server to server, without any sort of re-negotiation. The problem with RMI case, is much deeper than just a different *ObjectIdentifier*. In order to facilitate garbage collection, even though the only references to objects can be remote, RMI also has a distributed garbage collection that periodically checks to see if remote nodes are still alive and if they still have references to the server objects. This means that even if we are able to write the *ObjectIdentifier* as a client switched servers, it would also be necessary to somehow “transfer” the remote reference count from one server’s garbage collector to another. In short, the design of RMI, and the range of uses of RMI, make it impossible to make it a completely CORE-NS compliant agent.

There is also another problem since the response from the server would still be broadcast to all the clients that are linked to the server’s output stream, causing

the clients to receive data meant for the other clients, perhaps causing them to crash. One way around this is for links to be created and removed for each request/response. More appropriately, it shows that this peer connection implementation is severely inept for handling request/response systems including RMI, and any form of RPC for that matter. This dilemma is discussed with more detail in the conclusion.

Chapter 5

Analysis and Conclusion

In designing a generalized device interconnect and creating the three applications, it became apparent that the benefits of the peer connection paradigm are true, but that there are limits to which applications are suitable for using this paradigm. This section explores those limitations, and poses possible solutions, along with future avenues of work.

5.1 Analysis of Porting VNC

The most prevalent problem encountered while porting RMI and VNC to use CORE for network communications, is the fact these protocols were designed to separate connections between each client and server. VNC, for example, seems to be the perfect candidate for porting to CORE since it is easy to visualize the VNC server broadcasting the remote desktop to the multiple viewers, and that a viewer can be connected one of many VNC desktops, sort of like a TV(VNC viewer) tuning to different channels(VNC servers).

5.1.1 Access Control

The first thing that had to be dealt with was the connection setup phase which included the client authentication. While there is no analog in the CORE-NS, the

proper way of accomplishing this should be moving the access control into CORE itself. It is unknown how the security model should behave for peer connections, for example, whether the access control lists specify which agents are allowed to connect to an agent, or which third party agents are allowed to modify an agents connections. Security for the field of pervasive computing is still being heavily studied and will be a hot topic for years to come.

5.1.2 Automatic Translation

The greatest difficulty in porting VNC, is the parts of the RFB protocol which allows for the viewer to determine how the server sends out data. With TV's and TV stations, there is obviously no way for a TV to change the way a TV station transmits it feed. One solution is to simply make the viewer be responsible for accepting whatever pixel format the server sends out, though this could make the viewer too bulky for thin clients. Another solution is to make all the data going over the network conform to the same pixel format, even if this means that both the VNC server and viewers have the same native pixel format, but must translate to the set predetermined pixel format. Both of these solutions have their drawbacks, which is why both VNC and X have a negotiation phase. In CORE's case though, there can be a third solution – other agents can exist who's purpose is to translate one pixel format to another. So, when a linking agent connects a VNC server to a viewer, it also connects the proper middle translation agents together. Each of these middle agents can translate between a certain set of pixel formats. A system of automatic data conversion by passing through any number of operators was explored by the Ninja Paths[4] project. CORE has all the necessary parts – the name-specifier that describes the agent, and the ability to create links (paths) from agent to agent(operator). This means that something like the Ninja Paths project can simply be a linking agent in CORE.

5.2 Analysis of Porting RMI

It is not a surprise that the port of RMI did not have sufficient functionality to allow clients to switch servers, given that the design of CORE tries to prevent per client specific state. The problem, described in Section 4.3, is related to figuring out a way of controlling RMI's built-in methods of reference tracking. This reference tracking is related to remote references and the distributed garbage collection that comes along with it. This is a problem that is not inherent to RPC; it is only present when there are objects that are created and passed around the network. It might even be possible to make distributed garbage collection to work, but this would require extensive modifications to the functioning of RMI.

After all this, there is still the problem of the request/response nature of the protocol. Namely, that after the method invocation, the result, if any, is returned. In the current implementation of CORE, this is a significant problem, because that response will be sent to all the clients that are connected to the output stream of the RMI remote object server. This is a problem that, though not inherent to RPC, is inherent when using RPC on a function that returns results. That is, it is possible to create an RPC protocol which does not send responses, but such a protocol would prevent the use of functions that return data, greatly reducing the applicability of the RPC. Specifically, it would reduce PRC to purely a remote control mechanism – a client making the RPC server change its state.

Remote control mechanisms are common in pervasive computing – imagine if all appliances and devices around had embedded computers and were network enabled. Most of the communication to these devices would simply be to make it change its state: turn on light, change radio to station X, turn up the volume on TV, set oven temperature to Y, et cetera. Alas, it is also the case that there is generally a need for a request/response: what is the current oven temperature?, how loud is the TV?, is the door open?

5.2.1 Output Response Stream

One solution to the request/response problem is to differentiate the connections on an agent's output stream between a general connection, and a response only connection. The idea is that when a linking agent connects agent A's output stream to agent B's input stream, the linking agent must specify whether agent B will receive all the output of agent A, or simply the output of A that was generated as a response to some data from agent B. The method can be generalized such that the linking agent can dictate that a link only applies to responses meant for node B, C, and D.

This type solution does loosen the paradigm of peer connection a bit by introducing the concept of sessions (each request and response). It is possible to reduce the amount this peer connection paradigm is affected by providing a restricted API: an agent that receives a request does not know the origin of the message, but when the agent is sending the response, it states which message this is a response to. It is then up to the agent stub to figure out which agent the response is destined for.

Given this feature, it would be necessary to port RMI to CORE in a different manner. In the current port, the RMI server has one single connection to CORE, which is used to simulate multiple connections. The custom socket factories provide the `InputStream` and `OutputStream` interface to this connection. However, in order for the output response stream to work, it would be necessary for the interface to provide some way of indicating which message the response is intended for. The solution is to create a general TCP multiplexer agent.

The TCP multiplexer agent has one connection to CORE, and multiplexes into multiple TCP connections to a designated server, one for each request that the agent receives¹. Though the version presented multiplexes the single connection to CORE into separate real TCP connections, it is just as possible to do this purely in software like that presented in the original port of RMI, described in Section 4.3. Having this generalized TCP multiplexor will make it easier to port other applications that have

¹CORE provides its own form of data framing separate from the underlying network. This means that it is possible to aggregate a request into a single CORE message, even if it requires several TCP/IP packets to send. See Orlando Leon's thesis[12] for more detail.

a request and response nature, such as other RPC protocols.

5.3 Conclusion

This thesis project illustrated the benefits of a peer connection paradigm, but showed how, in its current state, it is not universally applicable. Perhaps the most important lesson is that there are different archetypical applications for the client/server versus peer connection. The main dividing line is whether or not the server keeps state about each individual client. In the case of RMI, this state is a necessity due to its goals and applications. With VNC, on the other hand, it is possible to remove all of this per-client specific state.

As shown in Chapters 2 and 3, the number of pervasive computing devices are growing, and that they benefit from the peer connection model. The question, then, is how to create services for this new paradigm, and how to adapt existing applications and services to this paradigm. There are several proposed new features that the Oxygen Research Group is exploring. The first three of these have already been presented along with suggestions on how to implement them in CORE-NS. Ultimately, the success of the peer connection paradigm depends on the success of implementing these features.

- Access specifiers – The ability to restrict which agents can connect to which agents.
- Automatic service composition – Automatically create links through a set of agents, that accomplishes a desired task, such as data translation.
- Output response stream – A third stream that contains the output from an resource agent that is intended only for the client.
- Delegation – Ability to have multiple CORE-NS's that share some set of their agents for use by agents in the another CORE-NS.

- Persistent Storage – A storage service that is distributed, fault-tolerant, and is persistent across re-starts. It is intended for all services to store all its hard-state.

Though this thesis did not completely answer this question, it does demonstrate that the peer connection paradigm can be extended much further than it is employed in resource managers like Rascal, or data translation like in Ninja Paths. Perhaps, above all, it demonstrates a need to explore this idea further, both in analyzing the issues raised in this paper but also for studying other benefits and applications of this paradigm.

Bibliography

- [1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Operating Systems Review*, volume 34, pages 186–201. ACM Symposium on Operating System Principles, December 1999.
- [2] Anoto group. <http://www.anoto.com/>.
- [3] Christian Bettstetter and Christoph Renner. A comparison of service discovery protocols and implementation of the Service Location Protocol. In *Proceedings of the Sixth EUNICE Open European Summer School: Innovative Internet Applications*, Twente, Netherlands, 2000. EUNICE 2000.
- [4] Sirish Chandrasekaran, Samuel Madden, and Mihut Ionescu. Ninja Paths: An architecture for composing services over Wide Area Networks. CS262 class project writeup, 1999.
- [5] Christopher Dabrowski and Kevin Mills. Analyzing properties and behavior of service discovery protocols using an architecture-based approach. Draft copy, June 2001.
- [6] Krzysztof Gajos. Rascal - a resource manager for multi agent system in smart spaces. In *Proceedings of The Second International Workshop of Central and Eastern Europe on Multi-Agent Systems CEEMAS*, 2001.
- [7] Arne Hess. How Pocket PC “talks” with a cell phone. *Club Pocket PC*, 1996. <http://www.microsoft.com/mobile/pocketpc/columns/ppcomm.asp>.

- [8] Javaspaces technology. <http://java.sun.com/products/javaspaces/>.
- [9] JINI network technology architectural overview. <http://www.sun.com/jini/whitepapers/architecture.html>.
- [10] Project jxta. <http://www.jxta.org/>.
- [11] Project jxta: Technical shell overview, April 2001. <http://www.jxta.org/project/www/docs/TechShell0verview.pdf>.
- [12] Orlando Leon. An extensible communication oriented routing environment. Master's thesis, Massachusetts Institute of Technology, 2002.
- [13] Jeremy Lilley. Scalability in an intentional naming system. Master's thesis, Massachusetts Institute of Technology, 2000.
- [14] Robert E. McGrath, M. Dennis Mickunas, and Roy H. Campbell. Semantic discovery for ubiquitous computing.
- [15] Robert E. McGrath and M. Dennis Mickunas. An object-oriented framework for smart spaces.
- [16] Sun Microsystems. RFC 1057: RPC: Remote Procedure Call Protocol specification version 2, June 1988.
- [17] Sun Microsystems. RFC 1094: NFS: Network File System protocol specification, March 1989.
- [18] Brenton A. Philips. Metaglu: A programming language for multi-agent systems. Master's thesis, Massachusetts Institute of Technology, 1999.
- [19] J. Postel. RFC 805: Computer mail meeting notes, February 1982.
- [20] Tristan Richardson and Kenneth R. Wood. *The RFB Protocol*. ORL, Cambridge, version 3.3 edition, July 1998.
- [21] Robert W. Scheifler. *X Window System Protocol*. Laboratory for Computer Science, Massachusetts Institute of Technology, version 11, release 6 edition.

- [22] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th International Conference on Mobile Computing and Networking (MobiCom)*, 2000.
- [23] Speech and Language Systems. Galaxy. <http://www.sls.lcs.mit.edu/sls/technologies/galaxy.shtml>.
- [24] Speech and Language Systems. *SpeechBuilder*. <http://www.sls.lcs.mit.edu/sls/technologies/speechbuilder.html>.
- [25] Sun Microsystems. *JDK 1.4 API RMI Documentation*.
- [26] Sun Microsystems. *Jini Technology 1.2.1 API Documentation*. <http://java.sun.com/products/jini/1.2.1/docs/api/>.
- [27] Sun Microsystems. *JavaSpaces Service Specifications Version 1.1*, October 2000.
- [28] Sun Microsystems. *JXTA v1.0 Protocols Specifications*, June 2001.
- [29] Sun Microsystems. *Applying the Factory Pattern to RMI*, 2002.
- [30] Sun Microsystems. *Java Remote Method Invocation Specification*, revision 1.8, java 2 sdk, standard edition 1.4 edition, 2002.
- [31] Sun Microsystems. *rmiregistry - The Java Remote Object Registry*, 2002.
- [32] Sun Microsystems. *Using a Custom RMI Socket Factory*, 2002.
- [33] VNC Virtual Network Computing. <http://www.uk.research.att.com/vnc/>.
- [34] Nullsoft developer network. <http://www.winamp.com/nsdn/>.

Appendix A

Complete code for Winamp Agent

```
package oxygen.core.example.winamp;

import ins.namespace.NameSpecifier;
import java.io.IOException;
import java.net.UnknownHostException;
import oxygen.core.agent.Agent;

public class WinampAgent
{
    private Agent agent;
    private Control control;

    public WinampAgent()
    {
        this.agent = new Agent();
        init();
    }

    private void init()
    {
        this.agent.setNameSpecifier(new NameSpecifier(
```

10

20


```

"[Device=Winamp][core=true[uid="+
    this.agent.getUid()+"]]");
    this.control = new Control();
}

public void run()
    throws UnknownHostException, IOException
{
    this.agent.connectAndRun();

    while(true)
    {
        String command = new String(
this.agent.receiveAndBlock());
        logger.debug("Received command: "+command);

        if (command.equalsIgnoreCase("play"))
            this.control.play();
        if (command.equalsIgnoreCase("previous"))
            this.control.previous();
        if (command.equalsIgnoreCase("pause"))
            this.control.pause();
        if (command.equalsIgnoreCase("stop"))
            this.control.stop();
        if (command.equalsIgnoreCase("next"))
            this.control.next();
        if (command.equalsIgnoreCase("title"))
            this.agent.sendString(
this.control.getSongTitle());
    }
}

```

```
public static void main(String[] args)
    throws UnknownHostException, IOException
{
    WinampAgent winamp = new WinampAgent();
    winamp.run();
}
}
```

60