Extending CORE for Real World Appliances
by
Glenn Eguchi

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 21, 2003

Copyright 2003 Glenn Eguchi. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by

Larry Rudolph
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Extending CORE for Real World Appliances
by
Glenn Eguchi

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Communication Oriented Routing Environment (CORE) is a pervasive computing environment designed to simplify device interconnection. CORE presents the abstractions of appliances, devices or software agents which can provide input and output streams, and links, connections between two appliances. This thesis uses CORE to present a design pattern for pervasive applications, a methodology for interfacing dynamic connections with hard state protocols, and a fault detection and prevention mechanism for pervasive applications.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist
Acknowledgements

I would like to thank Larry Rudolph, Jonathan Brunsman, Sonia Garg, Hana Kim, Arjun Narayanswamy, Kan Liu, Frances Wong, Shizuko Eguchi, and David Eguchi, without whom this thesis would not be possible.
# Table of Contents

Acknowledgements.......................................................................................................................... 5
Table of Contents.............................................................................................................................. 6
1 Introduction...................................................................................................................................... 1
2 CORE ............................................................................................................................................... 3
   2.1 Design Goals .......................................................................................................................... 3
   2.2 Components of CORE ............................................................................................................ 4
      2.2.1 Appliances ................................................................................................................... 5
      2.2.2 Cores ............................................................................................................................. 6
   2.3 CORE as a Pervasive Computing Environment ......................................................................... 8
   2.4 Implementation ...................................................................................................................... 8
      2.4.1 Wireless Keyboard ........................................................................................................ 9
      2.4.2 VNC .......................................................................................................................... 10
3 Problems with CORE .................................................................................................................... 15
   3.1 Link Problems ....................................................................................................................... 15
   3.2 Link Switch Problem ............................................................................................................. 15
      3.2.1 Hard State Protocol Incompatibility ............................................................................. 16
   3.3 Appliance Interaction .............................................................................................................. 17
4 Modifications to CORE ................................................................................................................ 19
   4.1 Core-Aware Appliances ........................................................................................................ 19
   4.2 Implementation ..................................................................................................................... 20
   4.3 Core-Unaware Appliances .................................................................................................... 24
5 Debugging Extension to CORE .................................................................................................... 26
   5.1 Reactive Warnings ................................................................................................................ 27
   5.2 Preventative Warnings .......................................................................................................... 28
6 Conclusion ...................................................................................................................................... 30
   6.1 Related Work ......................................................................................................................... 30
      6.1.1 INS ................................................................................................................................. 30
      6.1.2 O2S ............................................................................................................................. 31
      6.1.3 Service-oriented Network Sockets ................................................................................. 31
   6.2 Future Work .......................................................................................................................... 32
   6.3 Final Remarks ....................................................................................................................... 33
References .......................................................................................................................................... 35
1 Introduction

The Communication Oriented Routing Environment (CORE) is a pervasive computing environment designed to simplify device interconnection. This thesis uses CORE to present:

1) A design pattern for pervasive applications

2) A methodology for making dynamic connections compatible with hard state protocols.

3) A fault detection and prevention mechanism for pervasive applications.

CORE provides primarily two services: an application level routing environment and a device discovery system. CORE presents us with the abstractions of appliances and links. An appliance is a device or software agent which can provide an input and output streams. Appliances are connected to a core, a local machine with abundant computing resources. A link instructs a core to forward the output of one appliance to another. CORE additionally provides a basic naming and lookup system that allows users to find appliances based upon query criteria.

By combining the application level routing environment and the device discovery system, CORE can serve as a platform upon which complex pervasive applications can be built. This thesis clearly defines the role of an appliance in a pervasive application, and presents a general model for how pervasive applications should be built.

In many pervasive systems, the traditional connection abstraction is modified to accommodate the dynamic resources of the environment. For example, a pervasive DVD application may wish to play a DVD on the display closest to a user. As the user moves around the house, the destination for the DVD output changes frequently. CORE chooses to address this dynamicism by modifying connections to allow them to have dynamic destinations. However, many existing protocols utilize hard state; they assume that all data
previously sent over a connection was received by the user. Since this assumption no longer holds true for dynamic connections, hard state protocols pose a problem for them. This thesis addresses dynamic connections in the context of CORE, and proposes modifications to the appliance and connection to address the issue.

In pervasive computing environments, it is not clear that designing systems in an end to end manner [17] is the best design methodology, since individual devices 1) act in an independent manner and 2) do not always have the resources available to perform complex end tasks. Since all data in a CORE application passes through a core, CORE opens the opportunity for useful data stream processing. One possible usage is towards the debugging of pervasive applications. Pervasive applications are composed of a large number of devices, yet most pervasive systems do not address how to diagnose what went wrong if an error. This thesis extends CORE to allow fault detection and prevention.

This thesis is organized as follows: Section 2 presents a description of the original design of CORE. Section 3 describes problems discovered in the original design of CORE. Section 4 presents modified connection and appliance abstractions to allow a clear design pattern for pervasive applications and address issues arising from dynamic connections. Section 5 outlines an extension to CORE which enables fault detection and prevention. Section 6 presents related work. Section 7 proposes areas of future work. Section 8 presents conclusions drawn from this work.
2 CORE

The Communication Oriented Routing Environment (CORE) is a pervasive computing environment designed to provide a platform upon which pervasive applications can be easily built. CORE intends to simplify device interconnection, maximizing compatibility, and increase fault tolerance in pervasive applications. To meet these goals, CORE provides two services: an application level routing environment and a device discovery system.

This section presents an overview of the design goals of CORE, a description of CORE, and a discussion of implementation.

2.1 Design Goals

In the design process for CORE, the following goals were considered:

1) **Device interconnection should be easy.**

   The system should present an abstraction which is simple enough for a non-technical end user to understand. After a short explanation, a non-technical person should be able to connect devices (e.g. a laptop to a projector) using the system. However, the abstraction should be powerful enough to allow complex pervasive applications.

2) **The system should provide robust, flexible device naming and lookup functionality.**

   The system should allow users to specify descriptions for devices (naming) as well as query for devices that match high level descriptions (lookup). Additionally, the device descriptions should be flexible; users should also be able to customize the description of a device. Finally, the naming system must be robust. If a single device or server fails, the failure should be kept as localized as possible.
3) **Making devices compatible with the system should be easy.**

Many pervasive systems fail because they require either custom hardware or too much software. The system should minimize barriers to adoption by being compatible with a wide range of existing devices. Ideally, all that should be required of a device is network connectivity and if necessary, a simple wrapper.

4) **The system should increase robustness of pervasive applications.**

The system should provide some level of fault tolerance, or at a minimum, fault detection. The failure of an individual device within a pervasive application should be detectable. The failure of a pervasive application should have a minimal effect on other applications running using system.

The issues of security, privacy, and mobility were not addressed in our initial design. In the design process, the issues were discussed but concrete solutions were left to future work.

2.2 **Components of CORE**

We developed the Communication Oriented Routing Environment (CORE) to meet the above goals. As presented here, CORE leaves open the fourth goal of fault detection for pervasive applications. This problem was considered in the design for CORE, but will be addressed in significant detail later in this thesis.

One of the benefits of CORE is its conceptual simplicity. A CORE network is comprised of **appliances**, **connections**, **rules**, **cores**, and **messages**. The two fundamental building components of a CORE network are **appliances** and **cores**.
Figure 1: An illustration of CORE components. Several appliances are connected to a core: a software appliance (a VNC client), a hardware appliance (a cellular phone), and a core-aware appliance (the webcam). The core-aware appliance transmits data messages and control messages across its connection. The outgoing data messages are forwarded over a link to the cellular phone. The core is connected to other cores. Collectively the cores form a CORE network.

2.2.1 Appliances

An appliance is either a physical device or a software application that can provide connections to a core. A connection is a pair of input and output streams. Examples of connections include TCP sockets, serial connections, as well as local connections to the standard IO of a program. An appliance is expected to send and receive its relevant data over a connection. Examples of appliances include web servers as well as internet enabled speakers. An internet enabled speaker appliance would be expected to transmit sound data over its connections, while the web server appliance would execute http sessions over its connections.
Two types of messages are received over a connection. Control messages are messages intended for the core itself. A control message is preceded by a header which indicates to the core that a specified number of bytes following the header are intended as an instruction to core. Control messages are interleaved in the stream for a connection. This decision was made to encourage compatibility. Data messages are any data which arrives over a connection that is not a control message. An appliance that can send and receive control messages is called a core-aware appliance.

Notice that the requirements of the appliance abstraction are broad enough to encompass a wide range of devices. This generality was deliberate in order to encourage easy adoption of CORE. In order to support CORE, a device would simply need to accept connections on a TCP port. In practice, this generality would prove detrimental. Problems with the appliance abstraction are discussed in more detail in Section 3.

2.2.2 Cores

A core is an instance of the CORE program (For the remainder of our discussion, the lowercase “core” will refer to an instance of the CORE program, while the uppercase “CORE” will refer to the entire system). The machine running a core is assumed to have a wealth of computational power, memory, and bandwidth. In practical deployment, a core would be run on a desktop computer which has wireless capabilities and is connected to the internet. A core provides primarily two services to appliances:

1) **A core serves as an application level router for appliance data.**

   Within each core resides a number of links. A link can be thought of as a construct similar to the UNIX pipe. If a link exists from connection A to connection B, data messages received on connection A will be forwarded to connection B. Control messages are not forwarded. A core-aware appliance may create a link by sending an AddLink control message. Similarly, links may be removed by sending a RemoveLink control message.
Suppose we have the following situation: Alice would like to give a presentation which involves both DVD output and lecture slides from her laptop. In the CORE setting, the presentation room would have a core locally available and the projector, Alice's laptop, and the DVD player would be connected to the core. To display her lecture slides, Alice can create a link from her laptop to her projector in order to redirect her laptop video output to the projector. When she would like to display her DVD output, she can remove the link from her laptop to the projector and create a link from the DVD to the projector. As seen in this example, links provide a simple abstraction to allow the interconnection of devices.

Semantically, a link is a special case of a core rule. A rule is predicate-consequence pair executed by a core. Links are processed separately for performance reasons. Work is currently in progress to convert CORE as an entirely rule and attribute based system [2]. In the context of the discussion for this thesis, this distinction is not important.

2) A core provides device discovery capabilities.

The core for a lecture hall may have the digital projector and audio capabilities of the room connected to it, while the core for a person's office may have their desktop display and speakers connected to it. In order to allow devices connected to one CORE to be used elsewhere, cores are interconnected to form a device discovery overlay network.

Each core provides a basic device naming system in which it associates attributes with appliances connected to it. These attributes are arranged hierarchically. An appliance may associate an attribute with a device using the AddAttribute control message. Similarly, attributes may be removed using the RemoveAttribute control message.
Each core also provides support for device lookup. An appliance may send a Lookup control message to its core in order to find devices matching a particular set of attributes. Device lookup is currently implemented in a naïve manner. Lookup requests are flooded throughout the CORE network. If a core finds that one of its devices matches the requested attributes, it replies directly to the original requester with information about the device.

2.3 CORE as a Pervasive Computing Environment

The CORE system should be viewed a platform for pervasive applications. CORE adds the appliance and pervasive application layers to the traditional network application stack. A pervasive application manages the interconnection components on the appliance level by using links and lookup.

2.4 Implementation

An initial implementation of CORE was written in Java. Connections in the implementation of CORE were restricted to TCP sockets. Additionally, two test appliances were implemented for the initial design of CORE: Wireless Keyboard and VNC. This section describes these appliances as well as lessons learned from their implementation.
2.4.1 Wireless Keyboard

The Wireless Keyboard appliance demonstrates the usefulness of CORE for applications that generate relatively small amount of bursty traffic. The Wireless Keyboard appliance is composed of a single client that communicates keyboard input generated by a wireless keyboard to many servers. In this way the user can move around from machine to machine and control the input to each through the CORE network.

The Wireless Keyboard appliance is setup as follows. First, a wireless keyboard transmits keystroke data through infrared to a receiving computer on which the Wireless Keyboard Client (WKC) runs. This client connects to the CORE network and requests connections to one or more Wireless Keyboard Servers (WKS); that is, any device with the WIRELESS-KEYBOARD-SERVER attribute. These devices each run a server application that is connected to the CORE network and interfaces directly with the keyboard-event queue of their local machine. When a human user types using the wireless keyboard, the keystroke data is transmitted to the WKC via infrared, then to CORE across the network, and finally to a single WKS where it is inserted into the keyboard-event queue. Users control which WKS receives the keyboard data through the keyboard itself; that is, they can switch the CORE links by pressing preset keys.

The Wireless Keyboard Application generates relatively low traffic since keystroke data can be compressed to a very small size. Furthermore, the traffic is limited by its very nature since it is constrained by the typing speed of its human user. Finally, the traffic has been experimentally observed to be relatively bursty. Given these three characteristics, the Wireless Keyboard application performs exceedingly well within the CORE network. After timing 5,740 keystrokes, the average delay from transmission from the WKC to a core to the WKS was 4.479 milliseconds. The average network transmission time from the WKC to the core was 345 microseconds and the average time from the core to the WKS was approximately 425 microseconds. Thus, the transmission time from the WKC to the core to
the WKS (without being processed by the core) was 345 + 425 = 770 microseconds. The total time spent processing within the core was thus 4.479 – 0.770 = 3.709 milliseconds. The average transmission time from the WKC directly to the WKS was approximately 190 microseconds.

It is apparent from this data that the delay through a core for applications such as the Wireless Keyboard is extremely small. Although the use of the CORE network significantly increased transit time in this instance, it is impractical to expect that most network delays between hosts will be as small as 190 microseconds. Naturally, as network transmission time increases, the time spent within the core will become a significantly smaller percentage of the overall transit time. Thus, the usefulness of CORE depends upon the ratio of the delay of the host-to-host route versus that of the host-to-core-to-host route. This ratio will vary based on network congestion, bandwidth, and core processing capabilities. Appliances must therefore choose to sacrifice time for the other benefits that CORE provides.

2.4.2 VNC

Virtual Network Computing (VNC) [14, 19] is a typical client-server application that allows a user to view and control a remote desktop over a TCP socket connection. To test compatibility and explore the performance of CORE, we implemented wrappers for both the VNC Server and VNC Client for use in the CORE network. We chose to implement VNC for two reasons. First, the application is commonly used in well-connected networks. That is, VNC works best over high-bandwidth, consistently stable connections. Second, VNC's purpose is very similar to that of CORE: to allow devices to interact with each other remotely with as little effort as possible. The CORE-enabled VNC application allows users to switch the VNC server communicating to the client at will and with very little effort. Most importantly, however, the VNC application revealed several key issues for work with CORE.
Figure 3: The VNC application setup. Two VNC Viewer Clients, a VNC Server, and a Commander appliance are connected to a single core. The Commander adds and removes links when the user wishes to switch displays.

The VNC application is comprised of three components: the server, any number of clients, and the commander. The server and clients are wrappers for the actual VNC Server and Viewer programs (the need for these wrappers is discussed below, since one of the design goals of CORE was to avoid the use of wrappers entirely). The commander is a core-aware appliance that manages the links between the server’s and clients’ connections to a core.

The VNC application is setup as follows. Initially, the commander is manually connected to a core (or connects via device discovery). The commander then sends a series of add-connection control messages instructing the core to connect to the server and various clients. Next, the commander sets up a bidirectional link between the server and one client (the link is bidirectional since the VNC protocol operates on a client push model). Data then flows from the server to the core, through the link, and out towards the client. Finally, when
the user wants to switch clients, the VNC commander notifies the server and client wrappers and then sends a RemoveLink control message to the core, followed by an AddLink control message to create the link from the server to the new client. Figure 3 shows the setup and configuration of the VNC application.

Before discussing the performance of the VNC application, it is important to discuss the need for wrappers despite the desire to avoid their use. While initially CORE was designed to maximize compatibility with existing applications, wrappers nevertheless became necessary for the client and server. The wrappers primarily dealt with two significant problems that were unforeseen in the initial design of CORE: the hard state protocol and the link switching problems.

Unlike a traditional network connection, a CORE connection does not have a fixed destination throughout its lifetime. Links may be added or removed without notifying the affected appliances. This proves to be a problem for protocols which depend upon initialization when a new connection is formed. Like many protocols, the VNC protocol requires a handshake at the beginning of a new connection. Without the wrappers, the core would switch links without properly initializing the various VNC servers. The client and server wrappers are thus used to modify the handshake so that it occurs whenever the commander announces that a new link is created. This issue will be addressed in more detail in Section 3.

Additionally, the VNC appliance revealed a problem with the connection-as-a-stream abstraction of the original CORE design. While this abstraction greatly simplifies the system conceptually, CORE is unaware of where an application data unit begins or ends in an incoming data stream. Thus, if the source is transmitting data, a new link may be created in the middle of an application data unit. This causes the new destination to perceive the middle of a data unit as the start of the data unit, leading to corruption that is solely the
fault of CORE. To solve this problem in the VNC application, the server and clients notify the commander when it is safe to create a new link. In essence, they rely on their own knowledge of their application data units to refine CORE's AddLink and RemoveLink instructions. This issue will also be addressed in more detail in Section 3.

The VNC application also demonstrates another weakness of CORE: performance. Since all traffic between the client and servers must travel through a core, a significant amount of overhead is incurred. In their current implementation, the VNC wrappers require several seconds per screen update. While the CORE implementation can and has been optimized further, the bandwidth and delay overhead a core imposes will always be present unless a significant design change to include quality-of-service is made.

Performance testing indicates that a VNC client-server pair using CORE performs an order of magnitude worse than a VNC setup that does not use CORE. Performance data was collected through the following experimental setup: a server and a single client core were initialized in "direct" and "core" configurations. The "direct" configuration consisted of the server and client directly connected to each other across a high-bandwidth, low-latency TCP connection. The "core" configuration consisted of the server and client communicating through a core, also connected via high-bandwidth, low-latency TCP connections. The server and client programs ran on a single machine in order to guarantee clock synchronization, while the core ran on a separate machine. In order to ensure consistently heavy traffic, thereby simulating heavy load, the VNC wrappers were modified to request 100 full screen refreshes were performed.

Refresh time and throughput for connections through a core and through direct connections were measured.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Direct</th>
<th>CORE</th>
<th>CORE versus Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh Time (s)</td>
<td>2.91</td>
<td>22.3</td>
<td>766%</td>
</tr>
<tr>
<td>Throughput (kb/s)</td>
<td>166.7</td>
<td>21.6</td>
<td>13.0%</td>
</tr>
</tbody>
</table>

Table 1: VNC performance results.
refreshes requiring 480,016 bytes each. The send and arrival times of refreshes were logged and the results are summarized in Table 1.

The results show that there is a single order of magnitude difference between the performance of a connection through CORE and the performance of a direct connection. Since the tests were performed over a low-delay network at a time in which traffic and jitter were also low, the discrepancy can only be attributed to the computational overhead incurred by the core during data forwarding.

The VNC application demonstrates the success of CORE as an application-level router, but also highlights many failures that must be addressed before CORE is practically useful.
3 Problems with CORE

As demonstrated by the VNC appliance, CORE has several issues which must be addressed before CORE is practically useful. This section outlines these problems.

3.1 Link Problems

The presence of links caused several problems to arise in implementation of appliances for CORE. The first problem, the link switch problem, causes a data translation most of the time a link is switched. The second problem revealed was an incompatibility of CORE with hard state protocols.

3.2 Link Switch Problem

The low level link switch problem occurs when links are changed. The problem is best illustrated by an example (depicted in Figure 4):

A source appliance S and two sinks, sink A and sink B, are connected to a core. The data sent by S consists of series of two-byte shorts. In the current implementation of CORE, links are implemented by forwarding data one byte at a time. Initially, the source appliance S is linked to sink A. S begins sending shorts. A subsequently receives those shorts. At some point in the future, the user would like to switch the sink from sink A to sink B. The user sends the control message “remove-link S A” to the core. Since the core does not understand the underlying data stream of S, it arbitrarily decides where in the data stream to stop forwarding bytes from S to A. If it decides to stop in the middle of an application data unit (between two bytes of a short), sink A will be left waiting for input. This is not a serious issue since TCP packets are forwarded in a best effort fashion, and A should expect such delays.
Figure 4: The Link Switch Problem: We would like to redirect the output of a video source \( S \) from video sink \( A \) to video sink \( B \). The output of \( S \) is a series of two byte shorts. Initially \( S \) is linked to \( A \). After sending some data, the link is switched. Core decides to switch the link between the third and fourth bytes. When data forwarding resumes, \( B \) incorrectly interprets all subsequent data received.

The user then sends an “add-link \( S \) \( B \)” control message to the core. Again, since the core does not understand the underlying data stream, it arbitrarily decides where to begin forwarding bytes. If it decides to begin forwarding data in the middle of a data unit, all subsequent data received by \( B \) will be offset by some number of bytes. If \( B \) does not expect translation errors in its data (which it likely doesn’t since TCP is in order and reliable), \( B \) will interpret all subsequent data incorrectly or as corrupt.

This problem will affect most core-unaware appliances and will factor unnecessarily into the implementation of core-aware appliances. For longer data unit lengths, a bad choice of when to begin sending data becomes more likely. Specifically, for a data unit \( n \) bytes long, the probability core will split a data unit is \( (n-1)/n \).

3.2.1 Hard State Protocol Incompatibility

A higher level problem also arose in practice: CORE incompatibility with hard state protocols. In implementation of the VNC appliance, this problem arose in regards to initialization. While a traditional TCP socket consists of a static connection with a single partner, the CORE connection abstraction is slightly different. Since CORE introduces a
layer of indirection between two appliances, a sending appliance is no longer aware of the final destination of data it sends. The final destination is determined by the presence of links within CORE. More importantly, since links may be added or removed over time, an appliance cannot be guaranteed that it is communicating with the same appliance(s) throughout the lifetime of a connection. Due to this change in abstraction, interfacing some non-core aware appliances with CORE becomes a difficult task. This problem applies more generally to any system which modifies the end point of a connection to be dynamic.

Appliances with hard state are incompatible with the new connection abstraction. In the normal TCP model, one end of a socket can assume that the other end has received all data transmitted thus far. However, this assumption is no longer true in CORE. Therefore protocols which build upon this assumption may suffer from problems when interfaced with CORE. Generally any protocol with hard state, state which is not periodically refreshed (as opposed to soft state [3]) will suffer from some version of this problem with CORE.

Many protocols utilize hard state. In the above example, the VNC protocol does not send full screen refreshes, but instead a series of updates to the existing screen. Therefore, unless the entire screen is changed, the receiver's state will not be fully up to date. Other examples of stateful protocols include the telephony protocol H.323 [10] and X-windows [18]. Many protocols execute a handshake at the beginning of a connection to authenticate users. POP [8], SMTP [12], and FTP [11] are examples of such protocols. Thus, protocols with hard state are common and widely used and compatibility between hard state protocols and dynamic connections must be achieved.

3.3 Appliance Interaction

The original CORE presented us with an unclear model of appliance interaction. In UNIX, pipes provide a clear data path with the output of a series of pipes easily traceable back to the first component. CORE as is does not present a clear data flow across appliances. This is
due to inconsistencies between the appliance and link abstractions. The presence of
directional links conflicts with the bi-directional nature of core-unaware appliance
interaction.

Other elements of the appliance interaction were also left unspecified. In implementing an
appliance, it is unclear when a new connection to an appliance should be made and when an
existing connection be reused. The behavior of multiple incoming links to an appliance also
was not addressed.
4 Modifications to CORE

This section outlines modifications to the original CORE to address the link switching problem, the incompatibility with hard state protocols, and propose a clear model of appliance interaction.

4.1 Core-Aware Appliances

The cause of the problems described in Section 3 can largely be traced to the original appliance and stream abstractions. Thus to solve our problem, we shall redefine an appliance as follows: As in the original abstraction, an appliance is a physical device or software agent that can establish connections with a core. However, communication between appliances should be restricted to one way communication (except for possible feedback loops). An appliance can now be modeled as a filter, writing to an output stream based upon a set of input streams. This abstraction was intended in the original CORE, but never clearly outlined.

A web server maps nicely to the new abstraction. Its input should be web requests, while its output is web pages. So does an internet enabled monitor with its input being video data and its output being empty. However, peer to peer applications do not map nicely to the new abstraction since they generally require two-way communication.

A connection will also be redefined as a mapping of any number of input streams to at most one output stream. When combined with the above restriction of appliance interaction, a connection can be viewed as a session of appliance interaction. New connections to an appliance should be made when output only dependent upon a particular input are required.

The reason for this change is twofold. First, by clearly defining this role for an appliance we gain a clear notion of causality and the ability to exploit links more usefully. Directional links provide a clear data path for a pervasive application. If a device is generating incorrect
### Table 2: Control messages an appliance must support.

<table>
<thead>
<tr>
<th>Control message</th>
<th>Parameters</th>
<th>Return Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddDataConnection</td>
<td>None</td>
<td>Id of the output stream of the new connection</td>
<td>Creates a new data connection to the appliance</td>
</tr>
<tr>
<td>AddInput</td>
<td>Output Stream Id</td>
<td>Id of the new input stream</td>
<td>Creates a new input to a data-connection</td>
</tr>
<tr>
<td>AddOutputNotify</td>
<td>Output Stream Id</td>
<td>Position in data-connection of safe point to begin forwarding data</td>
<td>Notifies CORE of a safe place to add a new link to the output</td>
</tr>
</tbody>
</table>

output, incoming links to the device can be traced backwards to find the source of the problem. For example, if a speaker is playing incorrect output, we can check the mp3 player, microphone, and CD player inputs to see if they are functioning correctly.

Second, compatibility issues arose in making two-way communication between core-unaware appliances compatible with CORE. By limiting communication to one way communication, we can implement a generic wrapper for any core-unaware appliance.

### 4.2 Implementation

This section describes the implementation of the above abstraction changes and describes how these changes can be used to solve the problems encountered in the original design of CORE.

An appliance must support three control commands sent by core over a control connection: AddDataConnection, AddInput, and AddOutputNotify. Table 2 describes the parameters, return values, and effects of these functions.

Streams are assigned identifiers by CORE upon creation. This identifier is communicated to the appliance through a simple handshake process.
Core sends the AddDataConnection control message to an appliance when it would like to create a new connection to it. The appliance is expected to return an identifier to the output stream of the connection. An appliance creates an output stream by initiating a TCP socket to core, begins a new session, then sends the session output over the newly created output stream. Figure 7 illustrates this process.

If an appliance would like to refuse a new connection, it may do so by returning a failure code from the control message. In the appliance, AddDataConnection will typically be implemented as shown in Figure 6.

```c
Appliance::AddDataConnection()
01 Let socket = new TCP socket to core
02 Initiate new appliance connection
03 Begin sending connection output over socket.output
04 Return socket.output-id
```

Figure 6: Pseudocode implementation of AddDataConnection.
Core sends the AddInput control message to obtain a new input to a connection of an appliance. If the appliance would like to refuse a new input, it may do so by returning a failure code from the control message. In the appliance, AddInput handler is typically implemented as shown in Figure 8.

```
Appliance::AddInput(output-stream-id)
01 Let connection = connection whose output stream is output-stream-id
02 Let socket = new TCP socket to core
03 Interpret input from socket as an input to the connection
04 Return socket.input-id
```

Figure 8: Pseudocode implementation of AddInput.

To address the link switch and hard state problems, appliances must support the AddOutputNotify control command. Core sends the AddOutputNotify message to an appliance whenever a new link is added to the output of one of the appliance's connections. The appliance replies to the AddOutputNotify message with an index into the data stream from which the core can begin forwarding data. The first byte of a stream is index 0. Returning 0 from this function would replay the entire data stream. Returning infinity from this function would ensure that the link switch does not occur. Figure 9 illustrates the AddOutputNotify process.

By returning an index into the stream, we avoid the link switching problem. Notifying an appliance of when a new outgoing link is created allows appliances utilizing hard state protocols to refresh the state of receiving appliances accordingly. While there is some overhead incurred by this reinitialization, link switches are assumed to occur relatively infrequently. Thus the cumulative effect of the overhead is minimal. In the appliance, the AddOutputNotify handler is typically implemented as shown in Figure 10.
Core sends B an AddOutputNotify control message in response to an AddLink(B, C) control message.

B replies to the control message, and reinitializes the session after index 509.

When core reaches index 509, it creates the link from B to C.

Figure 9: Diagram of the AddOutputNotify process. Three appliances A, B, and C are depicted by squares. The core is depicted by the circle. Initially a link is present from B to C, and B is generating output data.

Appliance::AddOutputNotify(output-stream-id)
01 Let connection = connection whose output is output-stream-id
02 Finish sending data unit for session
03 Let index = the current index in session.output
04 Resend initialization data
05 Continue session
06 Return index

Figure 10: Pseudocode implementation of AddOutputNotify.

Core::AddLink(Connection S, Connection D)
01 if link(S, D) exists
02 return link(S, D).id
03 Let S_App = the source appliance
04 Let D_App = the destination appliance
05 Let input = D_App.addInput(D)
06 Let output_index = S_App.addOutputNotify
07 create link(S, D) and begin forwarding data from output_index
08 return the new link id

Figure 11: Pseudocode implementation of AddLink.
If the above control messages are supported in the appliance, a core should handle `AddLink` control messages as shown in Figure 11.

### 4.3 Core-Unaware Appliances

As stated above, only core-unaware appliances which utilize unidirectional communication should be interfaced with CORE. While this decision decreases the set of appliances which are compatible with CORE, this tradeoff is acceptable since the communication of such appliances would not map nicely to communication over unidirectional links. That being said, a generic wrapper for core-unaware appliances which utilize unidirectional communication can be written. Such a wrapper for an application which accepts TCP connections is presented in Figure 12.

```
CoreUnawareAppliance::AddDataConnection()
01 Let core_socket = new TCP socket to core
02 Let app_socket = new TCP socket to application
03 Forward all output of app_socket to core_socket
04 Return core_socket.output-id

CoreUnawareAppliance::AddInput(output-stream-id)
01 Let app_socket = socket with output id
  output-stream-id
02 If app_socket already has an input
   Fail
03 Else
05 Let core_socket = new TCP socket to core
06 Forward all input from core_socket to app_socket
07 Return core_socket.input-id
```

*Figure 12: Pseudocode implementation of `AddDataConnection` and `AddInput` for a generic wrapper of core-unaware applications.*

This wrapper makes CORE as transparent as possible in the connection; The connection through CORE using this wrapper closely resembles direct communication over TCP sockets. Since TCP sockets allow communication between exactly two parties, the above pseudocode allows exactly one input and one output for each connection.
How core-unaware appliances respond to link switches may consist of one of two generic responses. For some appliances, it may be appropriate to replay the entire output buffer whenever a new link is created from a connection. This corresponds to returning 0 from the AddLinkNotify control message. An example of an appliance where it is appropriate to replay the entire buffer is an internet radio appliance. If the radio appliance replays its entire output buffer, in the worst case, a receiving speaker appliance would replay all the radio output. However, if the outgoing data were time-stamped, the internet radio core-unaware appliance may appear largely identical to a core-aware appliance.

A second response a core-unaware appliance can have is to simply reject all link switches. This corresponds to returning infinity from AddLinkNotify.
5 Debugging Extension to CORE

While most pervasive systems provide naming, lookup, and device interconnection services [1, 4, 6, 7, 15], few provide fault tolerance or detection at the pervasive application layer. This section describes a debugging extension to CORE to allow basic debugging services at the system level.

Because of the large number of devices involved, isolating the problem in a pervasive application can be difficult. As an example, consider the following simple pervasive application:

Alice would like to watch her DVD as she performs some chores around the home. In various rooms of the home are display devices, internet enabled projectors, monitors, etc. Each display has a cricket beacon attached to it, and Alice carries a cricket listener and PDA around with her. As Alice's location changes the PDA reroutes her DVD output to the closest display device. When Alice is in the kitchen, the application works perfectly and displays her DVD on the display above her kitchen sink. However, when Alice moves to the bedroom, her DVD does not appear on the internet enabled television. What caused her switcher application to fail? Did the television lose internet connectivity? Is the network too congested? Did her cricket stop transmitting data?

Like most pervasive applications, Alice's switcher application involves a large number of components, and thus many possible points of failure. A realistic pervasive system should provide at least basic fault tolerance and detection in order to assist users and developers in diagnosing the problems in pervasive environments.

CORE naturally extends to assist debugging pervasive environments. All appliances data passes through CORE, CORE may act as a central point at problems can be identified. This section outlines modifications to CORE which allow reactive warnings, notification of
changes detected in a data stream, and preventative warnings, notification of potential incompatibilities of devices.

5.1 Reactive Warnings

In most pervasive applications, the majority of problems can be detected by examining rate statistics of the data streams. A connection upon which no data is sent or received is usually a sign of trouble. When using the Cricket Location Support System [13], one of the most common problems that occurs is that the cricket listeners or cricket beacons run out of batteries. In the data stream from the cricket daemon, this may be reflected as silence, or a lowered rate of transfer (e.g. if location information is only received from two beacons, when it should be received from four beacons). Another problem which commonly occurs in pervasive systems are network outages. Since pervasive applications make heavy use of wireless internet, easy detection of connectivity problems would be helpful in the debugging process. CORE eases debugging by logging rate statistics on all data streams. Currently burst lengths, interarrival times, and data transfer rates are logged. If a data rate deviates significantly from its average, CORE marks the deviation in a log and notifies the user of this deviation.

Change point detection can also be used with CORE to aid debugging of pervasive systems. The basic idea behind change point detection is to build up a characteristic of a data stream, then detect when the data stream deviates from that characteristic. One characteristic we have explored is constructing an approximate histogram of the values of incoming bytes. Algorithms for fast approximate histogram construction from data streams [4] are used to construct a characteristic of a stream over a fixed period of time (say a minute). If the histogram deviates significantly from previous histograms, this change can be flagged within CORE. Change point detection works for the less common error where the actual byte values
of a connection are invalid. Byte values may become invalid if a hardware device is not functioning correctly, or if a link is created between two incompatible devices.

Once suspect interactions are flagged, the task of pervasive debugging is simplified. As stated in Section 4, we can exploit the causal relationship of directional links to limit our search. Thus, we can begin at the output appliance (in the above example this would be the television), then trace backwards along incoming links. Any devices along the data path are likely to be problem appliances. The flagging mechanism restricts this set of devices making the task of debugging easier for both the end user and pervasive application developer.

5.2 Preventative Warnings

While the ability to link any two appliances is a benefit of CORE, this ability also allows links to be formed between incompatible appliances. To address this issue, we propose that CORE provide preventative warnings at link creation time. The basic idea behind preventative warnings is as follows: Over its lifetime, an appliance establishes many connections with a core. By examining the data streams of the appliance, CORE can build up a long term characteristic of the data stream. One such characteristic is a histogram of the data bytes over the lifetime of a connection. Another possible characteristic is average connection lifetime. As more connections are made with an appliance, CORE can formulate a more accurate characteristic of the appliance data stream. After a sufficient amount of time, CORE can use this characteristic in a preventative manner. For example, if the lifetime of the last 100 connections to an appliance have always been less than five minutes, it may be helpful in debugging to flag a connection which lasts longer than five minutes. A more rigorous analysis may involve the use of a histogram characteristic. Suppose approximate histograms for the input and output of two appliances have been generated over many connections. If the two appliances are compatible, the input histogram for the receiving appliance should resemble the output histogram of the sending appliance. Thus, if a link is
created between two appliances and their histograms deviate significantly, CORE could flag that the link may be a possible problem.
6 Conclusion

This section describes previous pervasive computing efforts similar to CORE, future areas of work for CORE, and final conclusions drawn from this research.

6.1 Related Work

This section describes other pervasive computing efforts similar to CORE. When applicable, the method by which each system addresses the dynamic connection issue and the proposed application model is compared against CORE's.

6.1.1 INS

The Intentional Naming System (INS) [1] is a resource discovery and location system for mobile and dynamic networks. INS integrates service discovery into the routing of a datagram across the network. Clients specify the destination of a datagram sent over INS by an intent, a set of attributes which describe the destination device. The datagram is then routed through an overlay network of intentional name resolvers and sent to any device which matches its intent. By late binding intents to locations, INS allows clients to communicate with services independent of their actual physical or network location. The advantage of such a scheme is that within a mobile or dynamic environment, services are constantly changing physical or network locations and by sending packets to name-specifiers, clients do not have to know how or where a particular service is connected.

INS suffers in that its communication is connectionless. Since binding of locations to intents is performed at delivery time, clients are unaware of the final destination of a datagram. Clients are not guaranteed that consecutive datagrams will reach the same destination. Therefore, hard state protocols will be incompatible with INS.
6.1.2 O2S

O2S [16] is a goal oriented pervasive system developed at the Oxygen Research Group [9]. A pervasive application in O2S is specified by a set of high level goals that must be met. Each goal can be satisfied by satisfying a set of sub-goals. Goals and sub-goals are recursively expanded to construct a goal tree. The O2S system then attempts to achieve the original set of goals by executing the steps required by a path through the goal tree. If any step in the path fails, new paths are attempted until the high level goal can be completed.

A goal ultimately resolves to the interconnection of pebbles. Pebbles are lightweight stream-based components similar to CORE appliances, and are connected through the use of connectors, a construct similar to CORE links. Connectors differ from links in that pebbles are connected in a distributed manner, and not through a central CORE.

While O2S provides a high level architecture for constructing pervasive applications, goals unnecessarily hide device interactions from the pervasive application. Since sub-goals are unaware of higher level goals, pebble connections created by lower level goals may unnecessarily conflict with higher level goals. Since connections are distributed, it would be difficult to identify such conflicts. Such conflicts would inevitably arise in practice should the depth of a goal tree become sufficiently deep. The create a new device in O2S, the device must not only adhere its first order goals, but also to a large number of higher order goals since these may fail due to ambiguities in specification. On the other hand, CORE exposes device interaction to the pervasive application, thereby allowing CORE applications to easily detect conflict. Finally, O2S requires an undefined high level heuristic to determine the best goal.

6.1.3 Service-oriented Network Sockets

Service-oriented Network Sockets (SoNS) [15] recognize the necessity of connection oriented communication. SoNS bind service discovery closely to the session layer, where the
destination of a service-socket is specified as a set of device criteria. A reusable planning layer continuously searches for devices which better meet the criteria. When a new device is found, the planning layer notifies the application using an application callback. At this time, the application may accept or reject the new choice of destination.

While SoNS provides connection oriented communication for a device, the policy of connection rebinding should be dictated at the pervasive application layer. Since it is the pervasive application that manages device interconnections, it is the pervasive application that should dictate policy of connection rebinding. If policy decisions are made at the session level, individual device policies may conflict with higher level application priorities.

6.2 Future Work

There remain many areas of active research for CORE. This section presents a description of five principle areas of work that can be explored: improved device lookup and discovery, domain specific cores, privacy, security, and mobility.

The device discovery and lookup system of CORE requires further work. Currently, CORE floods lookup requests across the CORE network. A more scalable method of lookup should be investigated. Additionally, many details regarding the attribute system have been implemented in a naive manner. Persistence of attributes, garbage collection of attributes, and a clear location for attribute storage have not been investigated thoroughly.

Additional benefits may be achieved by further relaxing the end-to-end restrictions on CORE. One possible application is the creation of domain specific cores. For example, a core could be restricted to use by video appliances. Such a core could provide increased debugging functionality and user utility. If the core understands the underlying data stream, it can examine the stream for possible errors more accurately. Additionally, the core can be more ambitious and correct recoverable errors. For example, if core detects that two appliances are
communicating using different protocols, core can translate one protocol into the other. This application of core as well as possible consequences should be explored in more detail.

Privacy is another possible application for CORE. If a core can be trusted, it can serve as a middleman for communication between two devices. The core can hide the identity of both ends of the connection from each other. This may be useful for applications like public displays in which a viewer of a public display may not want the owner of the public display to know his identity.

Finally, the issues of security and mobility were not addressed in the design of CORE. These issues must be addressed before CORE can be deployed on a larger scale.

6.3 Final Remarks
This thesis presented a filter-based design pattern for pervasive applications, implemented a methodology for interfacing dynamic connections with hard state protocols, and addressed the issues of fault detection and prevention in pervasive systems.

By combining an application level routing environment and the device discovery system, CORE can serve as a platform upon which complex pervasive applications can be built. This thesis clearly defined the role of an appliance in a CORE as a stream processing unit, and proposed that pervasive applications built upon CORE as the chaining of a collection of small modules.

In many pervasive systems, the traditional connection abstraction is modified to accommodate the dynamic resources of the environment. In the initial implementation of CORE, this modification caused the link switching problem and the hard state protocol incompatibility to surface. To solve these issues, we revised the connection and appliance abstractions and implemented an appliance notification scheme that allowed both core-aware and core-unaware appliances to safely communicate over dynamic connections.
In pervasive computing environments, it is not clear that designing systems in an end to end manner is the best design methodology, since individual devices 1) act in an independent manner and 2) do not always have the resources available to perform complex end tasks. This thesis explored the extension of CORE to facilitate debugging of pervasive applications through the usage of reactive and preventative warnings.
References


