The Application and Design of the Communication Oriented Routing Environment

by

Lawrence J. Brunsman

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 TEHCNOLOGY

May 21, 2003

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

The Application and Design of the Communication Oriented Routing Environment

by

Lawrence J. Brunsman

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 software-based, heterogeneous device interconnect system designed to facilitate interaction among devices in pervasive computing environments. The purpose of CORE is to establish an overlay network that allows devices and applications to interact while also providing developers tools with which they can monitor, configure, and restore system state. In doing so, CORE simultaneously functions as an application-level router and a debugging tool. As a router it inherits many challenges, such as traffic control, congestion, and quality-of-service. As a debugging tool it must provide users with the means to gain access to and modify the behavior of the system. The current implementation of CORE provides both of these functions as well as mechanisms for attributes, rollback, change-point detection, and user-defined rules.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist

Acknowledgements

I would like to thank my advisor, Dr. Larry Rudolph, for his inimitable guidance and help this past year.

I would like to thank the other members of the cast and crew of the Real World: LCS, including Glenn Eguchi, Sonia Garg, Hana Kim, Kan Liu, Arjun Narayanswamy, and Jorge Ortiz. They were all helpful, patient, and amusing friends.

I would like to thank my family, including my parents, Larry and Jackie Brunsman, and my grandfather, Harry Hawkey, for their continued support.

Table of Contents

1 Introduction

The Communication Oriented Routing Environment (CORE), a device interconnect system for pervasive environments, is designed to manage and simplify interaction between networked devices. CORE operates as a platform independent, application-level overlay network relying upon TCP [19] to communicate between devices. These devices require little or no modification to join the CORE network, and once joined, can take advantage of an array of tools, such as monitoring, configuring, and restoring the state of the networked environment in which they operate. CORE provides these tools by functioning as both an application-level router and a debugging tool capable of changing its behavior upon request from an attached device.

1.1 Motivation

The need for a system like CORE is easy to convey: pervasive computing has begun to live up to its name. Whether a laptop computer, handheld device, or smart phone, today's mobile computing devices are indeed pervading our everyday lives for the better. At least, they do so from the skewed view of the research laboratory while under the most scrutinized conditions. Outside the laboratory walls and convention showrooms, it is a rare sight to encounter the proverbial tourist asking his iPaq for directions or the smart shopper scanning barcodes into his Bluetooth keychain. What prevents the migration from theory to practicality for pervasive applications is not the lack of a killer application or weak demand; rather it is simply the difficulty to debug and use them.

Pervasive applications are difficult to debug because, for the most part, they rely upon dependability that developers have come to expect from the desktop computing environment. That dependability is neither present in today's mobile devices nor is it forthcoming. Additionally, developers have been slow to adapt to the varying requirements of mobile devices, choosing instead to focus on user interfaces and multimodal user input [10]. Rarely are applications developed to maximize battery life or support inconsistent network connections.

Pervasive applications are difficult to use because they simply are not robust. In fact, the old story of the tourist walking around an unfamiliar city with his iPaq needs to be updated. Before the tourist can find his way he must check the batteries on the iPaq and its sleeve; ensure the network card has been configured properly; move to a place where there is sufficient network reception; and most importantly, hope that the application and every application it interacts with are written with a degree of robustness far superior to that of the common desktop application. Who can blame the tourist for instead consulting a written guide when the use of persuasive devices is so arduous?

Clearly, the adoption of pervasive devices requires increased robustness and facility. To achieve these requirements, developers in turn need a tool to ease the difficulty of isolating and correcting errors that occur in pervasive systems. CORE was designed to do just that, allowing pervasive applications to bypass the endpoint-to-endpoint model that has proven to be unreliable and untenable in the mobile computing world. CORE's goal is succinct: ease the difficulty of using various pervasive devices by mitigating the complexity of device interaction.

The current implementation of CORE was built to improve the flexibility and robustness of previous systems by making its behavior customizable by users and developers. Built-in rules provide all the necessary functionality, while user-defined rules allow special behavior for those devices desiring more control. The goal of this implementation is equally succinct: ease the difficulty of debugging various pervasive devices.

1.2 Overview

This thesis describes the design, implementation, and performance of CORE. Chapter 2 provides an overview of related work and attempts to highlight the need for a system like CORE. Chapter 3 describes the design and includes a detailed description of the components of CORE. Chapter 4 continues with a description of the special functions of the current implementation, and Chapter 5 presents an analysis of its performance. Finally, Chapter 6 presents a discussion of unfinished work and conclusions.

2 Background Work

This chapter describes work related to CORE as well as a brief overview of its predecessors. The designs described span both software and hardware in various attempts to define the mechanics of device interaction. Again, it is this focus on implementation and a subsequent neglect of pragmatism that prevents these designs from ultimately achieving widespread use.

The initial purpose for a system like CORE [9, 12, 14] was to provide an overlay network for device interaction. Implementation focused on supporting Bluetooth-capable devices [18] rather than protocol-independent appliances. Furthermore, the system was not designed to mitigate the difficulties of debugging pervasive applications but instead focused on achieving meaningful communication through resource management. This approach – that resource management will serendipitously yield robustness and facility – is both a common and erroneous component of many pervasive systems' designs. Even worse, the implementation of the system suffered from poor scalability, complex compatibility requirements, and significant overhead to maintain the network. Consequently, that implementation was not used in this project but instead treated as a prototype with the ultimate goal of building upon its successes and avoiding its failures.

There are numerous other related projects that provide the infrastructure to unite various devices. For example, MetaGlue [5], described below, provides a sophisticated but complex framework for device interaction through extension of the Java programming language [17]. MetaGlue provides the infrastructure for both device interaction and resource discovery, while other systems, such as JNI [8], SLP [15], and INS [1], instead focus only upon the latter. Finally, there are countless projects that classify themselves as "ubicomp" or "interactive," such as the Interaction Spaces [21] project or the Intelligent Room [3]. All of these projects, however, share in the neglect to provide developers with sufficient tools to build effective and robust applications. The effort required to use these systems outside of a strictly controlled environment in many cases exceeds their utility.

MetaGlue, developed at the MIT Laboratory for Computer Science and Artificial Intelligence Laboratory, is a system that provides the infrastructure to maintain *intelligent environments* [5]. These environments, essentially collections of pervasive devices called *agents*, seek to bridge the physical and digital domains. For example, physical devices, such as televisions or lights, have virtual representations controlled by agents. These agents are implemented as an extension to the Java programming language and are maintained in a global SQL database. MetaGlue's design goals highlight the need for a distributed system with real-time response, knowledge of state and context, and support for dynamically changing hardware and software capabilities. Where MetaGlue fails is where CORE succeeds: MetaGlue's designers note the significant need for support for debugging yet have made little headway in achieving that need. Without that support, MetaGlue's immense capabilities will not be exploited outside of a laboratory.

Unlike the relatively high-level approach of MetaGlue, Universal Plug and Play (UPnP) [16, 20] allows device and service manufacturers to build their own APIs independent of language or platform. Service discovery is accomplished in UPnP using the Simple

Service Discovery Protocol (SSDP) [7], which uses XML for attribute storage. For a device to offer services, it must advertise its type of service and a URL address for its XML-based service description. This service advertisement can be done through IP multicast, allowing directory services and other clients to listen for the presence of devices. When a client needs a service, it multicasts a query request that can be responded to by either the provider or a directory index that knows of such a service. UPnP deserves credit for its robustness and usefulness, but it does not achieve these properties by providing extensive support for debugging. Rather, it limits the domain that engineers must consider: manufacturers need only provide the XML descriptions and APIs for their own devices.

Finally, the Intentional Naming System (INS) [1], closely related to CORE, is a service discovery and advertisement system. This project was designed as a resource discovery and location system for mobile and dynamic networks, thus providing greater flexibility than UPnP. INS consists of a network of Intentional Name Resolvers (INRs) that provides a resolution service for clients making requests. Devices can join the INS network by registering a name-specifier, which is essentially a description of the services it will offer, with a nearby INR. A client can then locate a general service by sending another name-specifier to an INR, which will return to the client a list of matching services.

INS differs from many service discovery protocols mainly in the way it describes services and routes requests: INS integrates service discovery with the actual routing of packets across the network. This integration allows clients to request and access services independent of physical or network location. In other words, INS allows clients to specify their intent rather than guessing where a service is located. The advantage of such a scheme is clear: within a mobile or dynamic network, services are constantly changing physical or network locations and by sending intentional packets to name-specifiers, clients need not know where a particular service is connected.

Service discovery, however, is neither CORE's primary goal nor its main benefit. Instead, CORE borrows many ideas from these related systems but derives most of its value from its simplicity and practicality. Independent of its service discovery mechanism, CORE provides users with the ability to debug their connections to discovered devices without having any prior knowledge of their address or location. This ability is provided through attribute-based addressing and tools like rollback and change-point detection.

3 Design of CORE

This chapter describes the design of CORE and the important features of its implementation. Section 3.1 describes the design goals of the CORE system, and Section 3.2 follows with an overview of its important components. Finally, Section 3.3 continues with a detailed description of the components of CORE.

3.1 Design Goals

The design of CORE is guided by four key requirements. First, the system must be simple to use by both end-users and developers. Details of device interaction – among both mobile and desktop devices – should be abstracted away from users and CORE itself should be transparent to most applications. Furthermore, existing applications should require little or no modification to be compatible with the CORE network.

Second, the system should be distributed and robust. Like any distributed system, failure of one or more components within CORE should not seriously degrade the performance of the rest of the system. Failures of nodes within the connected network are assumed to be more common within CORE than within standard overlay networks due to its distribution across both mobile and desktop platforms. Furthermore, since much of the complexity of device interaction is removed from the individual applications and assumed by CORE, many of the errors must also be assumed. Consequently, CORE must be robust in the presence of errors that occur both within CORE and within the applications that make use of it.

Third, the flow of data from a subset of connected components within CORE should be dynamically regulated by communication from another component. In other words, CORE should function as an application-level router that receives its forwarding instructions from other applications. The effect of this behavior is that the flow of data from one source can be directed to another source or sink by an independent third party. This effect is both beneficial – for example, multicast can be easily accomplished – and detrimental – a source may or may not have control over which or even how many applications will serve as its sink.

Finally, there must be no standardized naming system or hierarchy required by devices that connect to CORE. Any addressing scheme used by CORE and imposed upon other applications would subsequently require modification to those applications. Consequently, CORE must rely upon dynamic attribute caches through systems like INS.

3.2 Overview of CORE

A CORE overlay network is composed of one or more *cores*. A core is a software application that can:

Create and remove *connections* to *appliances*. Appliances are hardware devices or software applications that serve as the source of and sink for data transmission. Connections bind an appliance to a core and are independent of protocol; they are treated analogously to network flows or input-output streams. Appliances join the CORE network by creating a connection to a core, and leave the CORE network by closing that connection. Appliances may create multiple connections to a single core and may connect to multiple cores simultaneously. For example, a printer spooler is a software appliance that can connect to running cores through TCP sockets, allowing other appliances to make use of the printer.

Create, remove, and forward data across *links*. Links are internal constructs within a core and are similar to entries within a router's forwarding table: if a link exists between connections *X* and *Y*, any data that arrives from the input queue of connection *X* is summarily forwarded to the output queue of connection *Y*. Links are unidirectional and are not required to be symmetric. Multiple links can be forged between any two active connections.

Create and remove *attributes*. All cores contain a cache of attributes defined by name-value pairs. Attributes serve to name and describe the various appliances connected to a core and are used in a core's search mechanism. Attributes are added and removed through instructions from appliances and can be merged with the attribute cache of other cores. Attributes can be simple name-value pairs or hierarchically maintained as nested attributes, themselves composed of name-value pairs.

Figure 1 shows an example of two running cores and three connected appliances. The printer and mobile devices are each connected to a single core, while the desktop machine is connected to both cores. Data sent by the mobile device is forwarded via a link to the first core, where it is in turn forwarded to the printer. Data sent by the desktop machine to the first core is also forwarded to the printer, but data sent to the second core is dropped since there is no link. To the printer, the core is transparent since it does not distinguish between receiving data from the core or directly from the mobile device or desktop machine.

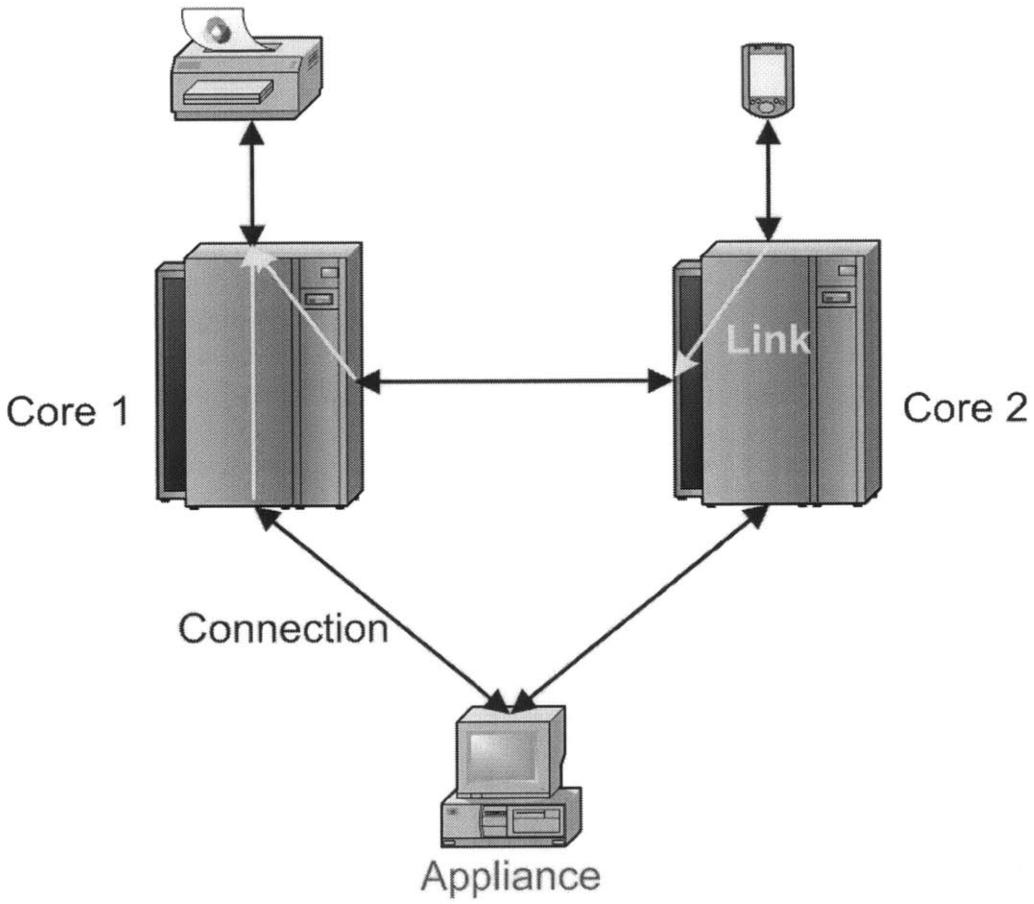

Figure 1: An example scenario using CORE.

3.3 Components of CORE

This section describes in detail the various components of CORE, as they are currently implemented, and further describes the interaction between these components.

3.3.1 The CORE Database

Within each core is a database that provides storage to encapsulate its complete state, including all connections, attributes, and recent events. Each core maintains a separate, independent database that may be replicated and restored at any point. The database includes state variables for each connection, such as their addresses and ports, attributes, and frequency of transmission. Furthermore, the database maintains a history of events that have occurred while the core has been running to provide support for rules and rollback.

The CORE database provides several advantages over a stateless core. First, debugging the system is greatly simplified since applications can observe the complete state of the core both before and after they change it. Second, maintaining a history of events provides the ability to undo changes that were detrimental or had unexpected results. Finally, the database allows for increased robustness in the unusual case that the core unexpectedly fails and must recover from an external source.

The CORE database is contained entirely within the memory of the running core's process and, unlike a SQL database such as that used in INS, cannot be accessed directly by other devices or processes.

3.3.2 Events

A CORE event is a wrapper for various system events that occur while a core is running. Events are generated when an appliance connected to a running core transmits data or instructions to the core, thereby altering its state. For example, a DATA-ARRIVAL event is generated whenever a core receives data from one of its connections. The data is wrapped into the newly created DATA-ARRIVAL event along with the identifier of the transmitting connection and other status parameters. The event is then inserted into the database to be used by the rule engine.

Events are purged from the database periodically, but this procedure occurs so infrequently that events, for all practical purposes, may be considered as existing forever within the core's database.

3.3.3 Rules

Rules in CORE allow users to define the behavior of the system without requiring them to modify code or use complex object-access protocols. Rules are composed of a predicate, consequent, and inverse. The predicate is a statement that evaluates to true or false and, upon evaluating to true, triggers the consequent to fire. The predicate can operate on any of the components of a core: connections, events, attributes, or even other rules. For example, the predicate of a Link Rule might test for the presence of a DATA-ARRIVAL event within the database from a specific, enabled connection.

A rule's consequent is an instruction that a core can interpret and carry out. A consequent may or may not alter the core database. For example, the consequent of a Link Rule would instruct the forwarding of data from one specific connection to the outgoing data buffer of another specific connection. The successful completion of a rule's consequent triggers an event notification that is stored within the core's database.

A rule's inverse is a unique feature to CORE which allows rules to be undone. The inverse essentially reverses any effects caused by the consequent of a fired rule. Thus, after applying a rule's inverse, the core cannot determine whether the rule ever fired or whether it was fired and then undone. For example, if a rule were fired to remove a link between two connections, that rule's inverse would be the restoration of the removed link.

There are four rules native to a core: *connect*, *link*, *attribute*, and *rollback.* The connect rule creates or removes a connection to a specified host on an indicated port. The link rule creates a link between two indicated connections. The attribute rule adds or removes an attribute to particular connection. Finally, the rollback rule initiates the rollback of any number of previously fired rules.

Figure 2 shows an overview of the rule engine and database as it applies a link rule. Upon arrival, data is wrapped into an event an inserted into the core's database. The rule engine then tests the predicate of each rule stored within the database and triggers any rules that can be fired. The result is the firing of two link rules, effecting the forwarding of data.

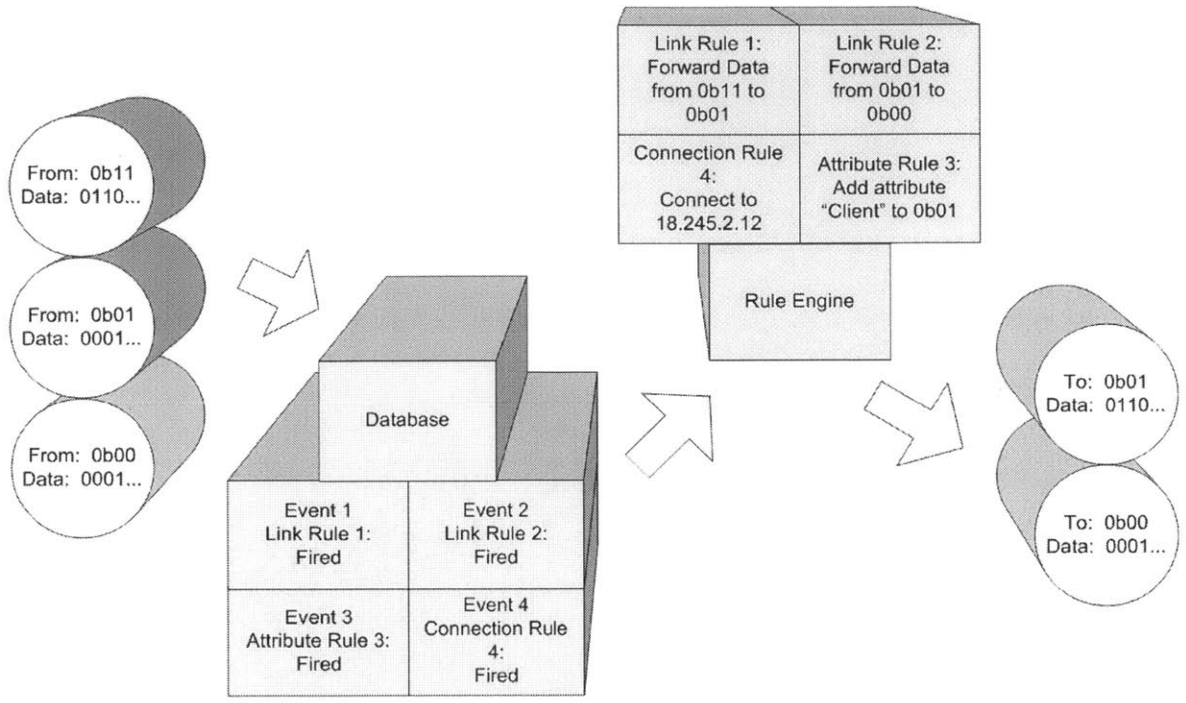

Figure 2: The rule engine architecture of CORE.

3.3.4 Connections

A core's connections are implemented as TCP sockets and are thus bidirectional and insecure. Connections have a limited state that includes a unique identifier, a status flag, a time-to-live value, and diagnostic variables. The status flag is used to mark connections as active or inactive independent of whether the underlying TCP socket is open or closed. The time-to-live value may be set so that, if no data is received or transmitted via that connection after the indicated time, the connection will be automatically closed. The diagnostic variables are used by a core to provide debugging tools to users and are described in detail in Chapter 4.

There are two types of connections: those that transmit *application data* and those that transmit *control data.* Application data is defined to be data that is sent across a connection to a core but is meant to be forwarded or ignored. That is, application data does not contain instructions for a core and appears identical to data that would be transmitted if the core were not present. Conversely, control data contains messages and instructions intended to be parsed and used by a core. A core will only generate responses to appliances that have forged a control connection, and thus the core is transparent to any appliance capable of using only application data.

3.3.5 Links

As described, links instruct the forwarding of data from one connection to another. Since links are applied so frequently, they are implemented outside the rule engine to improve performance. Their implementation is via a modified hashtable that permits multiple values per key. The table provides the ideal mechanism for links: fast lookups with very little unused space. On average, lookups require constant time and in the worst case require linear time with respect to the number of links created per key. Insertions into the link table are achieved in constant time, while deletions require linear time with respect to the number of links stored per key.

3.3.6 Attributes

Attributes within a core are based on those proposed in INS but are less rigidly defined. Attributes are name-value pairs that can be used to name or describe an appliance connected to a running core. For example, a printer spooler connected to a core might install a series of attributes to define its printer's capabilities.

Many systems' implementations of attributes – including previous versions of CORE – were too complicated to be of much use to applications unless they included a very robust attribute parser. These parsers were needed to interact with the complex hierarchy used to store and retrieve attributes. While conceptually a solid design, applications that rely on CORE should not be forced to use a computationally intensive attribute parser whenever they need to store simple attributes. Consequently, the standard hierarchical attribute structure was augmented with a simpler, more flexible structure. Attributes are not limited to pairs of names and values and are no longer required to be hierarchical (although the attribute author can impose a hierarchy, if so desired). Instead, attributes can be stored based entirely on the structure in which they are added.

Like INS, however, a core's attribute cache still supports the maintenance and use of a hierarchy for complex attributes. This bipartite implementation has both benefits and drawbacks. Its chief benefit is the allowance of simple, lightweight attributes for mobile devices. Its chief drawback is that the lack of complexity within simple attributes inhibits the ability to execute complex queries.

Figure 3 shows an example of the two types of attributes supported by CORE. The upper right part of the figure shows attributes for two appliances defined by name-value pairs that are stored as a list. Searching for a query using these attributes requires matching the query against the names of each attribute. Given *n* attributes, this search requires $O(n)$ time to visit each attribute, along with the time spent to search for the pattern within the string.

The bottom left part of the figure shows the hierarchical structure used by a core to store attributes similar to that of INS. Searching for an attribute is similar to traversing a tree: when the branching factor is smaller than the number of attributes, its performance is markedly better than the simpler list model. The drawback of using a hierarchical attribute structure is its complexity to use and maintain.

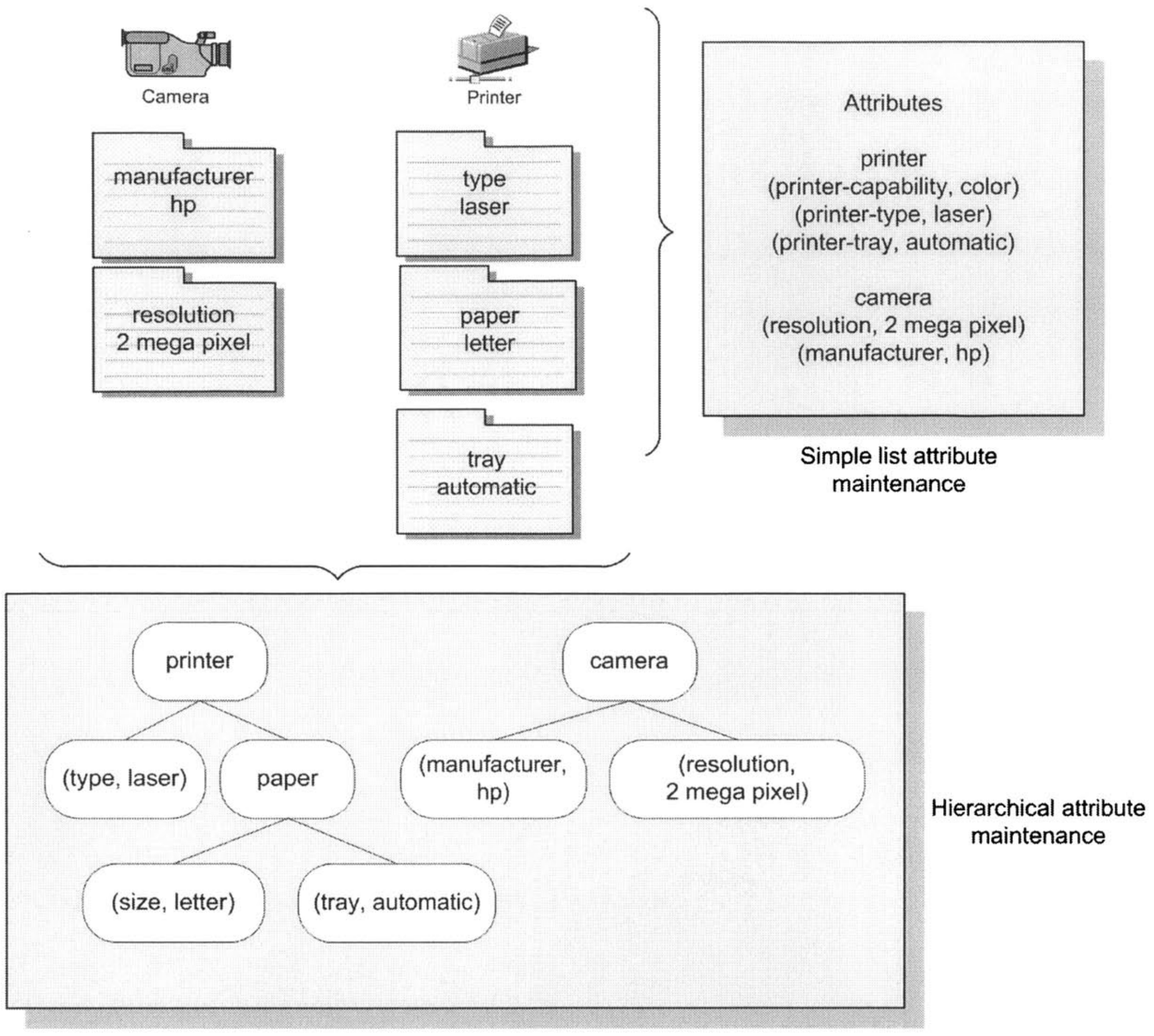

Figure 3: Two different ways to store attributes within a core.

Special attributes, such as those related to a core's implementation of connections – including host addresses, port numbers, and unique identifiers – are omitted from public view. While the core still maintains theses attributes, they are presented only upon request by an appliance for debugging information. In practice, this behavior greatly simplifies the interaction between applications and attributes. Finally, users can specifically instruct the addition of any of these special attributes into the public view of an appliance's attributes.

4 Special Functions of CORE

The current implementation of CORE provides several mechanisms, including rollback, change-point detection, and rules, that are specifically designed to overcome the limitations of previous versions. First, rollback provides a tool that effectively undoes a sequence of rules in order to rollback the current state of the system to a previous state. Second, change-point detection, a relatively new technique borrowed from other fields of study, uses observations of variables to aid in the diagnosis of errors. Finally, the rule engine provides support for users to define their own rules and events, thereby altering the behavior of the entire system if so desired. This chapter describes these special functions.

4.1 Rollback

The rollback functionality provided within CORE allows applications to undo a series of rules that may have caused unwanted side-effects. Applications can rollback any number of rules, which will sequentially be undone by firing their inverses.

The rollback process begins when an application sends a rollback control message that contains the *rollback number*, which specifies the number of rules to invert. The most recently fired rules are retrieved from the database in a last-in, first-out manner, similar to a stack. Next, an event is added to the database indicating the rollback process has begun. Finally, the rules are undone by sequentially firing their inverses, completing the rollback process.

Rollback within CORE is limited to the sequential inversion of its rules. That is, rollback will not undo changes that occur outside of the scope of the core. For example, suppose a chat application is connected to a core. Furthermore, suppose that a link is inadvertently switched and another user receives unintended messages due to the malformed link. The user can rollback that link and return the state of the system to the original link configuration, but cannot retrieve the data forwarded erroneously.

The usefulness of rollback unfortunately depends directly upon its complexity. For example, removing a mistakenly forged link through the use of rollback is convenient but does not fully demonstrate the power of the rollback functionality. Instead, its behavior is analogous to the "undo" command that users take advantage of daily. The undo command, while useful, is by no means capable of the benefits of a well-designed and well-implemented system restore. This feature of some databases and operating systems allows users to set checkpoints before making significant changes, allowing them to reverse those changes should they result in adverse effects. Analogously, as the undo feature is markedly simpler than the system restore, so too is the rollback feature within CORE. To achieve the richer benefits of system restore, users must define detailed inversions of their own rules in order to guide the correct behavior of the rule engine.

Rollback within CORE is also limited by its implementation. Each core's database maintains only one centralized event log for the entire system. Consequently, various connections can generate events that are then interdigitated within the event log.

Rollback is difficult to achieve because the desired events can be separated by an unknown number of unrelated events. This behavior cannot be corrected in the current implementation of CORE but is correctable by modifying the system. For example, the database can maintain an individual event log for each appliance, preventing events from being interleaved. The immediate drawback to this division is that rollback is limited to events that are generated only by a single appliance; it subsequently would not be able to modify the entire system.

Figure 4 shows an example of the CORE database during the rollback process. In step 1, the database contains events for three fired rules: an attribute rule, a link rule, and a connect rule. In step 2, a rollback rule arrives with a rollback number of two. After it is processed by the rule engine, three new events are added to the database. The first is an event indicating the firing of the newly arrived rollback rule, while the second and third events indicate the inversion of the connect and link rules, respectively.

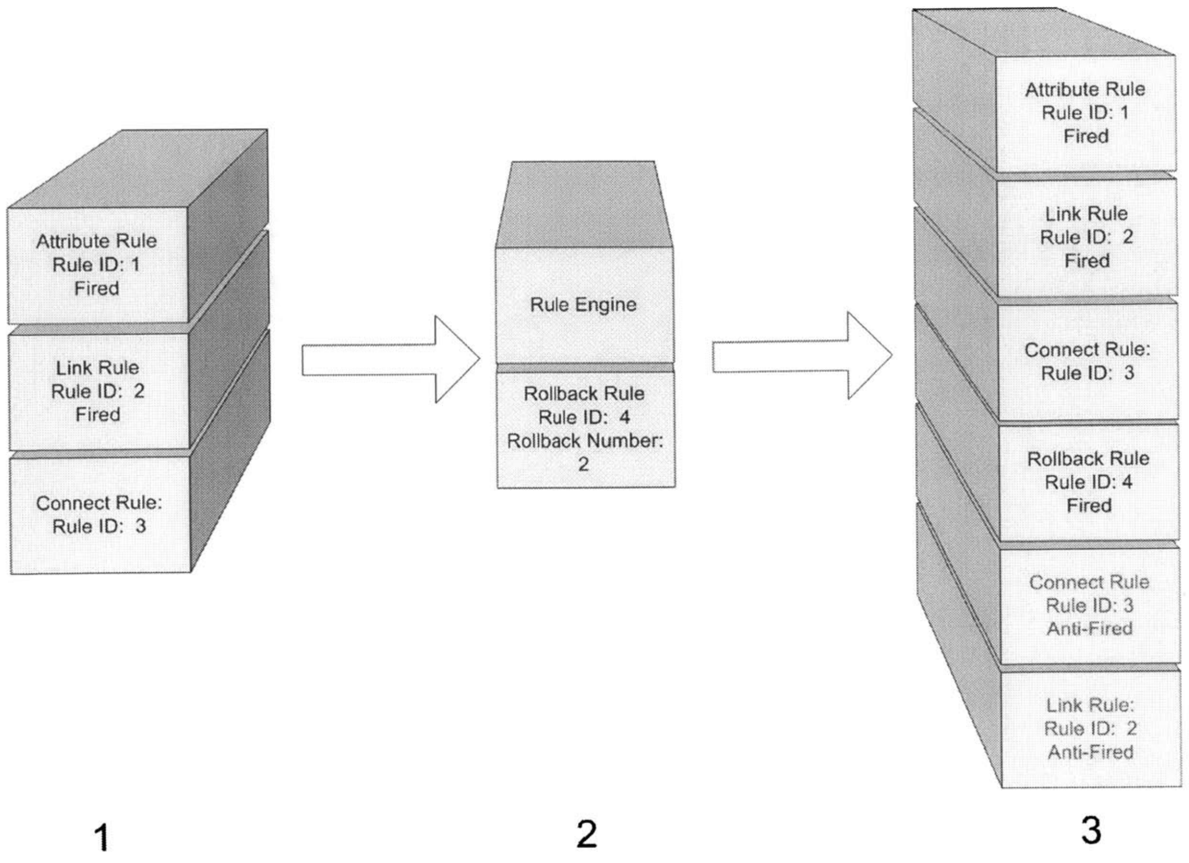

Figure 4: The rollback process's effects on a core's database.

4.2 Change-Point Detection

Change-point detection is a relatively new technique used to aid in the automatic detection of problems within running code. Change-point detection has long been employed as a tool in signal processing but is slightly modified in this context. A variable is marked to monitor closely and whenever that variable takes on an unusual value, a warning is issued. Over time, the accuracy of change-point detection increases since the accuracy of the probability distribution of the variable's possible values is also increased. Techniques such as Bayes Nets and Markov modeling lend themselves ideally to change-point detection since they attempt to predict unknown probabilities based on cause-effect relationships.

Change-point detection is included within CORE to monitor the input and output rates of connections. Whenever the ratio of the current transmission rate to the average transmission rate is extremely large or extremely small, a warning is generated. The current transmission rate is measured by averaging the transmission rates of the connection over the past ten seconds. The average transmission rate is measured by using a weighted infinite average.

Observation has shown that the strongest benefit of change-point detection within CORE is from the identification of failing connections. If a connection usually transmits consistently but then stops transmitting or becomes intermittent, a core's change-point detection would quickly identify that behavior as symptomatic of a possible problem. Consequently, a warning would be presented to the user indicating the presence of a possible failure in the connection.

Some connections, however, transmit sporadically and the change-point detection mechanism is of little help in diagnosing problems. Warnings generated by a core's change-point detection relevant to these types of connections are spurious. To avoid seeing these warnings, change-point detection within the core can be disabled on a per-connection basis through the use of attributes. Change-point detection is enabled on all connections by default.

4.3 User-Defined Rules

CORE allows users to customize its behavior by installing their own rules. There are no requirements placed upon user-defined rules, but the following subsections detail several issues that should be considered by rule authors.

4.3.1 Event Consumption

There are many events that trigger rules to fire but should do so only once. That is, once an instance of an event triggers a rule, that same instance should never retrigger the same rule. For example, when a DATA-ARRIVAL event occurs, it should only trigger a Link-Rule to fire once for each matching rule, thereby preventing the link from acting on old events. In other words, data should not be linked repeatedly if it has already been linked.

Since rules cannot alter events themselves, they must instead consume events by utilizing the event number, a unique identifier that is assigned to events upon insertion into a core's database. This identifier is guaranteed to be sequential and increasing, allowing rules to use a simple comparison when needed. The Link Rule, for example, never copies an event with a previously-seen event number. Although most rules will need to accommodate event consumption, it is important for rule authors to determine the individual appropriateness of event consumption for their own rules.

4.3.2 Behavior of Rules and Past Events

In the case that a rule is added after an event that satisfies its predicate, the rule author must dictate whether that past event should trigger the new rule. In some cases the rule should not distinguish between event arrival time and rule arrival time. For example, a Data Arrival event that occurred prior to a Link Rule's creation should not necessarily trigger that rule to fire. In other cases, prior events should trigger a rule's predicate. Consequently, rule authors must be careful to define the behavior of their rules with past events. The Link Rule, for example, specifies that prior DATA-ARRIVAL events do not satisfy the predicate.

4.3.3 Behavior of User-Defined Rules and Rollback

Rule authors may choose to include or omit an instruction to invert their rules. Rules that omit instructions on reversing their behavior will not be undone during a rollback. They will, however, be included in the rollback count. This behavior is adverse in that changes to the system will not be undone despite the intent of the rollback to do so. It is advised but not required that all rules include rollback instructions.

5 Performance of CORE

The performance of CORE was measured through controlled simulation of network traffic and system load. Four different sets of experiments were conducted to measure its performance under varying conditions. The following sections describe these experiments and their results.

Throughout the following discussion of performance, the term *load* will be used to refer to the number of connections simultaneously connected to a core. Thus, heavy load would indicate that many connections are actively transmitting data to the core, while light load would indicate the opposite. Additionally, the term *traffic* will indicate the quantity of data a connection transmits to the core. Heavy traffic indicates that a connection transmits a large amount of data, while light traffic indicates the opposite.

Finally, there has been some debate over the efficiency and performance of Java TCP-sockets compared with those of the C programming language. Since CORE was implemented in Java, the results of such debate are important to consider when examining its performance. Fortunately, the difference in the performance between the two languages' implementations appears to be negligible across the Internet, as shown in [11]. These results, however, may not translate well to a mobile network and must be treated with a degree of skepticism.

5.1 Experiment 1: Heavy Traffic with Light Load

The first experiment was conducted to measure the performance of a core with heavy traffic and under light load. In this experiment, one connection was established to the core to serve as a source of data, while a second connection was established to serve as the sink. A single link was forged between the two connections, and a large file containing 3,932,214 bytes was transmitted through the core. A special server was used to time the transmission process by noting the time at which the first byte was transmitted and the time at which the last byte was received. The times were measured to the nearest millisecond. A series of control trials were then performed with the same client-server pair transmitting and receiving the same file, but omitting the core. Figure 5 shows the results of the first experiment.

The average time required to complete the core's trials was approximately twice as long as the direct trial. The core's performance in this experiment suffers from the *transmission performance penalty*, which is charged to the CORE system because unlike direct client-server connections, each byte must be transferred twice (once to the core from the source, and once more from the core to the destination). This penalty cannot be avoided without the use of direct connections as a type of quality-of-service mechanism. This approach is discussed further in Section 6.1.3.

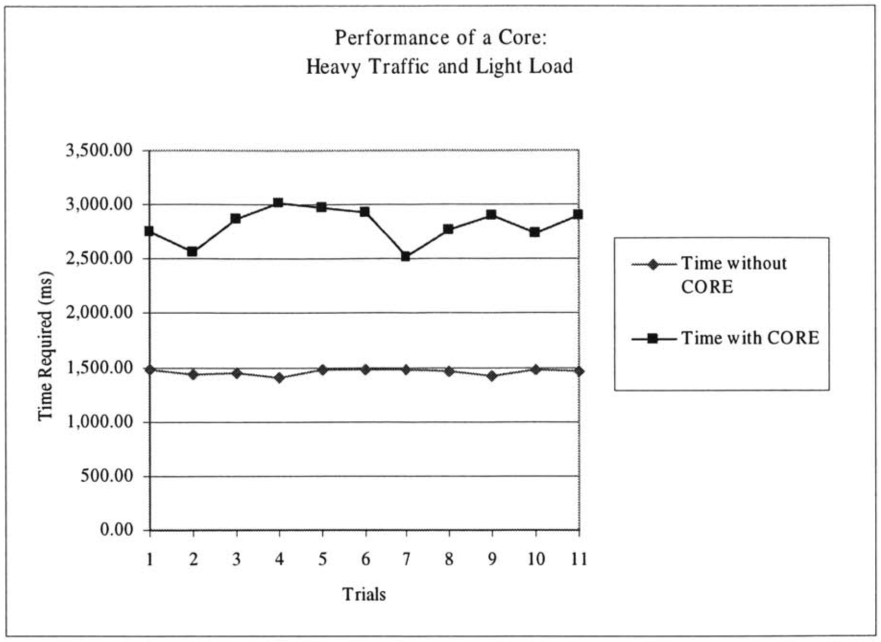

Figure 5: The performance of a core under light load but with heavy traffic.

5.2 Experiment 2: Heavy Traffic with Heavy Load

The second experiment was conducted to measure the performance of a core when large amounts of data were transferred under heavy load. In this experiment, four connections were created to simulate four different data sources. Each of the four connections was linked to another, unique receiving connection. The four connections then transmitted a large file containing 3,932,214 bytes to the core concurrently with a random variation of less than one second in their start times. Each of the four receiving connections was attached to a dedicated server that received the data from the core. After the final byte was received, the servers noted the time to the nearest millisecond. Finally, similar to the first experiment, control trials were performed by connecting the same client to the receiving server and then transmitting the same file, but omitting the core. Figure 6 shows the results of the second experiment.

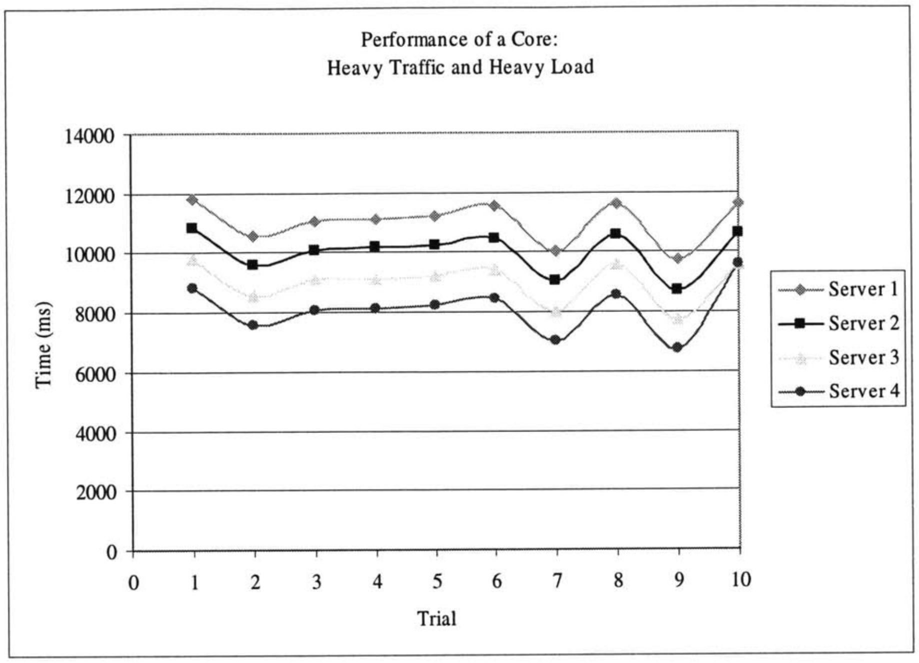

Figure 6: The performance of a core with heavy load and heavy traffic.

As expected, the time required to complete the transfer significantly increases, since CORE's linking mechanism is round-robin. This behavior demonstrates the fairness of such a strategy in that all four transfers require nearly the same amount of time. Also note that while the traffic has increased by a factor of four, the time required increased by slightly less than a factor of four.

5.3 Experiment 3: Light Traffic with Light Load

The third experiment was designed to gauge the performance of connections that do not transfer large amounts of data or do so sporadically. The same experimental setup was used as in the first experiment, except that the file size was reduced to 117,550 bytes.

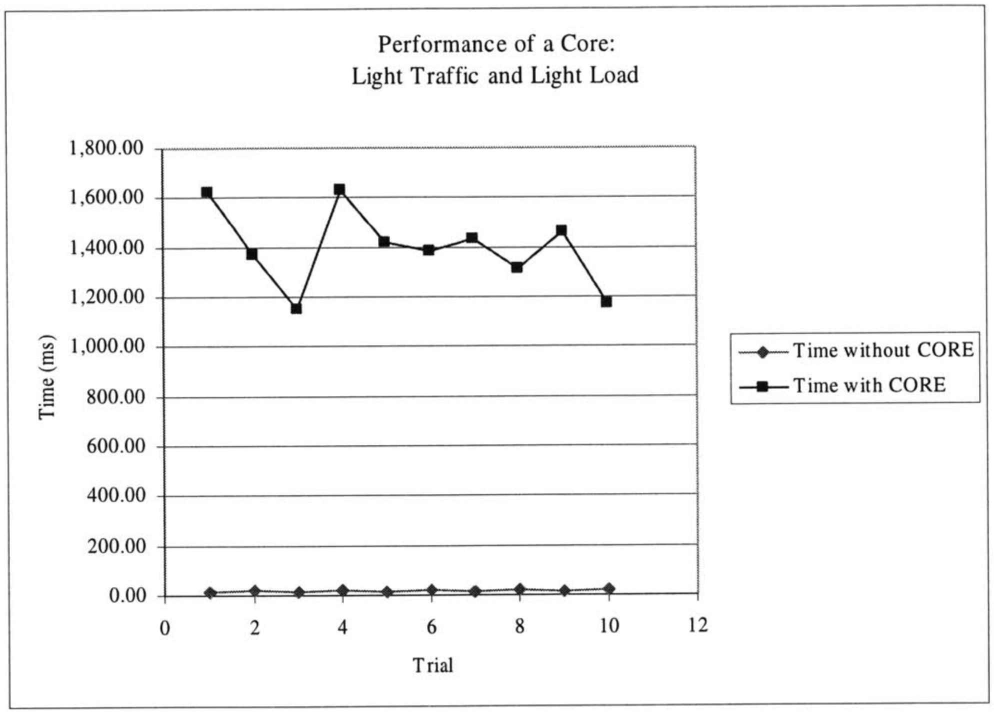

Figure 7: The performance of a core under light load and light traffic.

Figure 7 shows the results of the third experiment. The performance of the core, compared to the control, is significantly poorer than in the large-file transfer experiments. The chief cause for this degradation in performance is the *overhead performance penalty*. The penalty is charged to the maintenance and monitoring of the various connections required by the core as it runs. This performance penalty, unlike the transmission penalty, can be mitigated in several ways. These options are also discussed further in Section 6.1.4.

5.4 Experiment 4: Links

The final experiment was conducted to measure the performance of a core's link mechanism. This experiment measured the total time required to multicast a large file of size 3,932,214 bytes. Two servers were setup as dedicated listeners and attached to the core, while one client transmitted the file. Two links were forged connecting the client to both servers simultaneously. Figure 8 shows the performance of the core under this experiment.

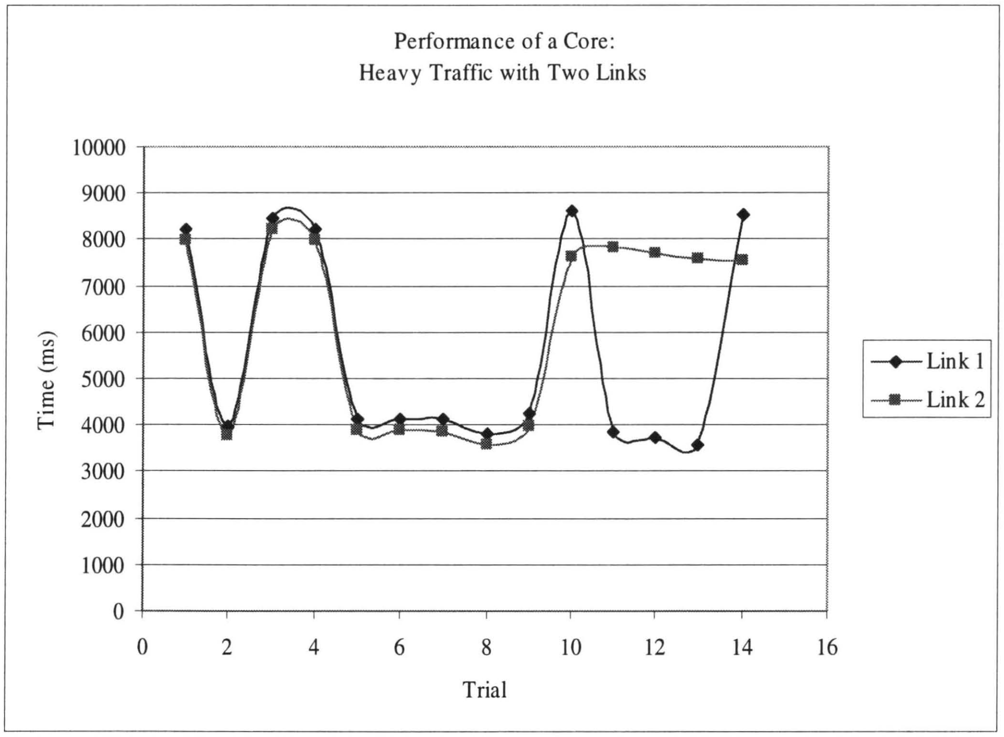

Figure 8: The performance of a core's link mechanism.

The results of the link experiment are difficult to interpret. The time required to complete the multicast transmission ranges from a minimum of approximately 4000 milliseconds to a maximum of nearly 8500 milliseconds. The expected time to complete the multicast is approximately 5000 to 6000 milliseconds, or twice the time required to unicast the same amount of data.

The wide disparity between the experimental trials is most likely due to CORE's implementation. A core is broken up into several running threads to monitor its various connections, the database, and the rule engine. Periodically, the database reaches its maximum storage quota and old events are discarded. This garbage collection process requires a significant amount of time – approximately one thousand milliseconds – during which no data is forwarded through the core. This particular delay could have occurred just before the second link was given control of the CPU and just after the first link ceded control of the CPU. Consequently, in some trials the transmission time between the two servers is markedly different.

6 Conclusion

This chapter presents the areas of research that remain open after the conclusion of work on CORE. It then presents conclusions from the design of CORE and its implementation.

6.1 Future Work

The previous designs of systems like CORE led directly to the work completed for this project, which included numerous improvements over its predecessors. Similarly, the design and implementation of CORE have led to several problems that can be addressed in future work.

6.1.1 Security

CORE is an insecure system that places no effort on providing users with privacy or security. An easy addition to this system is the incorporation of a class of service that includes a flag for secured communication. Connections can then be encrypted to prevent plaintext transmission of sensitive data. Access control lists can be used to manage creation and removal of links for specific connections.

Even with these enhancements, however, CORE remains an insecure system. A malicious user can flood the system with traffic or control messages, causing the equivalent of a denial-of-service attack. Significant design and implementation work remains necessary before CORE is deployable in an untrustworthy environment.

6.1.2 Link Breaking

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 devices that depend upon initialization when a new connection is formed. Without wrappers to modify such devices, a core could switch links in midstream without providing proper initialization.

Thus, while the connection abstraction used by CORE – treating connections as streams and disregarding their protocols and implementations – greatly simplifies the system conceptually, it increases the difficulty of changing links. CORE is unable to determine when it is safe to interrupt a sequence of data by removing an old link and creating a new one. Consequently, if a source is transmitting data, a new link may be created in the middle of an application's 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. Solving this problem without the use of wrappers remains an open question.

6.1.3 Rollback Boundaries

Additional improvements to the rollback mechanism can provide users with increased control and improve the ability to debug pervasive applications. Exploring the creation of rollback boundaries – divisions between rules and devices that can and cannot be rolled back – is necessary before a complex rollback can fully be achieved. For example, user defined rules that do not include rollback definitions or devices that cannot be rolled back can be segregated to a specific core.

6.1.4 Quality-of-Service

Quality-of-service algorithms were not included in CORE other than those mechanisms native to TCP. There is significant evidence indicating that QoS algorithms could vastly improve the performance of future implementations of CORE. For example, it seems wasteful that connections that communicate frequently through a stable link should have to suffer from the CORE bottleneck. Their performance would greatly increase if the communication between the two appliances was accomplished through a direct, point-to-point connection.

There has been some investigation already into the incorporation of QoS into CORE. Three classes of service for connections have been identified: a class composed of connections that are monitored by a CORE at all times, a class that is monitored very infrequently, and a class that bypasses CORE altogether. The first class represents that which is already implemented. The second class would be useful for connections that very infrequently transmit rules or requests. These connections should be checked only occasionally by CORE's connection management threads. Finally, the third class of service would be useful in situations like the example above in which connections desire fast, point-to-point communication.

Networking research has produced significant advances in QoS algorithms and their performance. This research should be reviewed in further detail and QoS should be implemented in the next version of CORE. Finally, fair queuing algorithms [6] should be investigated and possibly applied to a core's input buffers.

6.1.5 Limiting Overhead

The overhead required to maintain a core penalizes its performance, especially under light traffic (see Chapter 5). There are several ways to reduce this penalty. For example, implementation on a multiprocessor machine can reduce CPU overhead, while multiple network interfaces can reduce multicast delay.

A more theoretical approach to reduce the overhead performance penalty is to maintain links through the use of a Random Access Machine with Byte Overlap (RAMBO) [13]. This model of computation is similar to the familiar Random Access Machine but allows byte overlap in memory. For example, a shared matrix memory configuration on a RAMBO machine is setup such that both each row and each column represents a word in memory. The *i*th row overlaps with the *i*th bit of each column, allowing for constant time updates of multiple words; in other words, allowing for constant time updates of multiple connections' output queues in the running core.

6.1.6 Bayes Nets and Change-Point Detection

The change-point detection algorithms implemented within CORE are primitive yet they are also very effective. Nearly all problems with communication across pervasive systems involve the cessation or gradual slowdown of transmission from a particular device. This cessation or slowdown can be easily detected by the running core.

There remains a case, however, for enhancing the change-point detection algorithms used within a core. For example, if rollback boundaries are developed and implemented, change-point detection could be used to identify erroneous rollback subsets. This more sophisticated detection would require more sophisticated techniques. One such technique is the incorporation of Bayes Nets within the change-point detection mechanism. Bayes Nets provide a mathematical tool to predict events based on probabilistic modeling of causes and effects, seemingly an excellent fit for change-point detection. More work, however, is necessary to confirm this hypothesis.

6.2 Conclusions

Any system that attempts to connect to and interact with other devices on behalf of a third party must accept two tenets. First, as an application-level router, the system must be capable of handling or delegating the same tasks performed as a network router, including traffic control and congestion, quality-of-service, and security. Second, as an ad-hoc overlay network, the system must provide an easy and efficient way for appliances to join the network, to leave the network, and to carry out device discovery and lookup. Adding to these two requirements, any system that attempts to interact with pervasive devices must provide improved support for both software and hardware debugging through means of diagnostics and tools.

This thesis has presented the design and implementation of CORE, a system that attempts to satisfy these requirements. CORE makes pervasive systems easier to use by emphasizing simplicity and providing tools for debugging. Much work remains to be completed before CORE is usable in practice, including security and performance improvements. Ultimately, however, the benefits of a system like CORE will become increasingly important as more networked devices attempt to interact.

Pervasive systems, intelligent environments, interactive spaces, mobile computing, and ubiquitous computing are meaningless labels. The systems bearing these labels are often impractical and unwieldy, despite countless scenarios and demonstrations to the contrary. The solution to this disparity seems simple: for the time being, pervasive devices must become dependent on systems that provide debugging tools and must favor utility over performance.

7 References

[1]. Adjie-Winoto, W., Schwartz, E., Balakrishnan, H, Lilley, J. "The design and implementation of an intentional naming system." In *Proceedings of the 17th ACM SOSP*, Kiawah Island, SC, Dec. 1999. http://nms.lcs.mit.edu/projects/ins/.

[2]. Balakrishnan H., Kaashoek F., Karger D., Morris R., and Stoica, I. "Looking up data in P2P systems." In *Communications of the ACM*, February 2003. http://nms.lcs.mit.edu/6.829/p2p-lookups.ps.

[3]. Brooks, R. "The Intelligent Room Project." In *Proceedings of the 2nd International Cognitive Technology Conference*, Aizu, Japan, 1997.

[4]. Clark, D., and Tennenhouse, D. "Architectural consideration for a new generation of protocols." In *Proceedings of the ACM SIGCOMM*, Philadelphia, PA, September 1990. http://nms.lcs.mit.edu/6.829/alf.pdf.

[5]. Coen, M., Phillips, B, Warshawsky, N., Weisman, L., Peters, S., and Finin, P. "Meeting the Computational Needs of Intelligent Environments: The Metaglue System," in *1st International Workshop on Managing Interactions in Smart Environments.* December 1999: pp. 201-212.

[6]. Demers, A., Keshav, S., and Shenker, S. "Analysis and Simulation of a Fair Queueing Algorithm." In *Internetworking: Research and Experience*, Vol. 1, No. 1, pp. 3-26, 1990.

[7]. Goland, Y., Cai. T., Leach, P., Gu, Y., and Albright, S. "Simple service discovery Protocol." Internet draft. < http://search.ietf. org/internet-drafts/draft-cai-ssdp-v1-02.txt>.

[8]. Jini. <http://java.sun.com/products/jini>.

[9]. Kao, A. "Design and Implementation of a Generalized Device Interconnect." http://org.lcs.mit.edu/pubs/theses/akao/main.ps.

[10]. Kim, H. "Multimodal Animation Control. " Masters of Engineering Thesis.

[11]. Krintz, C., and Wolski, R. "Using JavaNWS to Compare C and Java TCP-Socket Performance." In *Journal of Concurrency and Computation: Practice and Experience,* Volume 13, Issue 8-9, pp. 815-859, 2001.

[12]. Leon, O. "An extensible communication-oriented routing environment for pervasive computing." http://org.lcs.mit.edu/pubs/theses/leon/leon_thesis.pdf.

[13]. Fredman, M., and Saks, M. "The cell probe complexity of dynamic data structures," in *Proceedings of the 21st ACM Symposium on Theory of Computing*, pp. 345-354, 1989.

[14]. Ortiz, J, and Kao, A. "Connection oriented routing environment: A generalized device interconnect." http://org.lcs.mit.edu/pubs/ortiz.pdf.

[15]. Perkins, C. "Service location protocol." http://playground.sun.com/srvloc./slp_white_paper.html.

[16]. Rekesh, J. "UPnP, Jini and Salutation - A look at some popular coordination framework for future network devices." Technical Report, California Software Lab, 1999. http://www.cswl.com/whiteppr/tech/upnp.html.

[17]. See http://java.sun.com.

[18]. "Specification of the Bluetooth system." http://www.bluetooth.com/pdf/Bluetooth_11_Specifications_Book.pdf.
[19]. *Transmission Control Protocol*, in RFC 0793. http://www.ietf.org/rfc/rfc0793.txt.
[20]. Universal plug and play specification. <http://www.upnp.com>.
[21]. Winograd, T. "Interaction Spaces for 21st Century Computing." To appear in *John Carroll, Ed., HCI in the New Millennium*, Addison Wesley. (in press)