An Ambient Computing System

by

Jesse M. Davis

B.S.C.S. University of Kansas, Lawrence Kansas 1998

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

Professor in Charge

Committee Members (2)

Date of Acceptance

Acknowledgements

To my friends and colleagues at ITTC, who have provided me with friendship and insight, not only into my thesis but also into myself.

To my parents, Ron and Helen, for their love and support through the years.

To Robin, who had changed my life for the better since the first moment I met her, and who continues to do so each day.

Abstract

An ambient computing environment coordinates a variety of computing and network-enabled devices to present a seamless, customizable and eventually invisible interface to the user. The idea of the meta-operating system applies the qualities of traditional operating systems to this environment. This allows devices and applications to be created and integrated much like they are in traditional operating systems. The meta-operating system also allows personalization, presence and permissions to be offered in a natural way, which has not previously been done. This thesis documents the architecture design, implementation and evaluation of a meta-operating system which provides these three services.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is an Ambient Computing System?

An ambient computing system is a software framework that coordinates a variety of computing and network-enabled devices in order to ease their use in home and business environments. A diverse set of computing, networking and software resources can then be controlled through one seamless, customizable interface. By simplifying setup and management of myriad devices, the entire system eases user interaction, allowing the system to become a pervasive, and eventually invisible, part of the user's environment.

1.2 Motivation

Traditional approaches to consolidate and control devices in a home or business environment have not provided the type of infrastructure that an ambient computing system does. Home automation systems with proprietary control software do not have

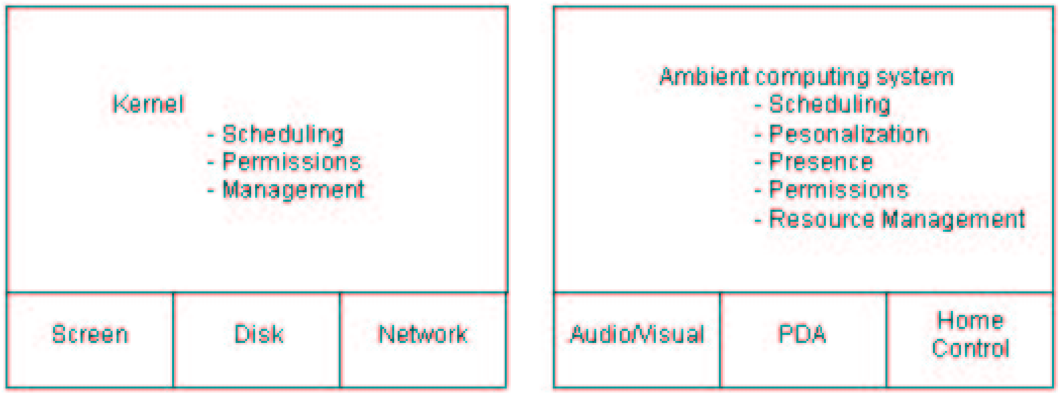

Figure 1.1: Operating System Analogy

the extensibility to handle the number of new devices that come to market. Products such as X10 modules [30] do not have capabilities for advanced features, such as supporting multiple users, which requires support for personalization and permissions.

The idea of applying operating system qualities to a distributed device architecture creates a software architecture capable of enabling many new services by providing a common way to integrate and coordinate devices and applications. Personalization, presence and permissions can then be incorporated into this architecture much like they are in a traditional operating system.

The system described in this document provides a software-based, flexible, easy to use computing and networking environment that is compatible with current and future devices. It embraces the idea of a meta-operating system, which applies the traditional definition of the operating system to a software system not dependent on hardware. This architecture incorporates the ideas of personalization, presence and permissions, which have not been implemented as a whole in previous systems.

1.3 Project Goals

This project hopes to:

- Define a system that integrates all network capable resources in an area into a easily controlled system.
- Provide the ability to trigger sets of actions to perform on the system when external stimuli from the environment are received.
- Implement the infrastructure necessary to personalize this system to the user's needs.
- Create an interface to this system so that integration with other pervasive computing systems will be possible and simple.
- Show the validity of the implementation by showing correct results from the execution of common operations on the system.

1.4 List of Accomplishments

The following was accomplished while implementing this system:

- A new architecture for an ambient computing system was designed that incorporated the ideas of personalization, presence and permissions.
- Events, actions and devices were defined.
- A set of XML messages to control the system was defined.
- A message transport architecture was written to relieve blocking and provide better scaling support.

- The database interface was created.
- The interface for personalization of any object in the system was provided.
- Support for macros was designed and implemented.
- A SOAP/XML-RPC interface was written to provide interoperability with other ambient computing systems.
- A Perl API was written to use the SOAP/XML-RPC interface, allowing devices to be written in Perl in addition to Java.

1.5 Layout

This document is organized into the following sections:

- Background – Background information needed to understand this document, as well as similar projects.
- Architecture – An explanation of the system architecture that specifies the control logic and hardware interaction of this ambient computing system.
- Implementation – A detailed description of the implementation of a software product that follows the requirements in the architecture system.
- Results – Description of the test environment and test results from using the implemented system.
- Conclusions and Future Work – Lessons learned and future work to complete on the system.

Chapter 2

Background

2.1 Vision

This work describes an architecture and implementation that realizes the goal of providing a software framework for an ambient computing environment. This environment ties together all network capable devices in a domain into a single intelligent entity. The foundation for this framework is the concept of a meta-operating system, which strives to provide many traditional operating system features to a diverse set of elements.

2.2 Supporting Technologies

2.2.1 Distributed Computing Technologies

Distributed computing seeks to increase computational power by creating architectures to solve problems in parallel. In terms of hardware, two approaches can be taken. On single machines, multiple processors can divide sections of tasks between them. Message passing and concurrency takes place quickly, since the memory cache is local. Threads can be used as a programming aid to simplify programming for multiple-processor machines, abstracting the division of tasks away from the programmer. However, contention for memory by the processes limits the number of processors that can be used.

To combat this, the second approach can be taken. Tasks can be divided between separate computers, each with its own memory and processor. A master node hands off sections of tasks to slave nodes, which process the sections and return the results to the master node. The processor and memory can be dedicated to this sections, alleviating memory contention. This approach also scales more readily than multiple-processor machines, and as proven by the Beowulf project [23], can be done cheaply with off-the-shelf PC components. Here, the problem is not of memory contention, but of efficiently sending messages between master and slave nodes.

Software architectures can then utilize these distributed hardware systems to provide large-scale computational environments, where attention must be paid to load balancing, resource control, and fault tolerance. These architectures might support thousands or millions of operations. One such architecture is the Common Object Request Broker Architecture (CORBA) [20]. Objects are only allowed to interact with each other through their specified interfaces. Implementation of objects are encapsulated and managed by the Object Request Broker (ORB). Multiple ORBs can be run on multiple machines, and objects can invoke methods on remote objects transparently. This is known as location transparency, one of the key requirements of a distributed computing architecture. The format of the remote request is handled by a common protocol used between ORBs.

However, the emphasis of an ambient computing environment is the coordination of processing instructions and communications between ad-hoc collections of heterogenous devices, or nodes, and not the distribution of computational power across those nodes. A CORBA implementation may be a bit heavyweight, especially in cases where memory usage is crucial, such as embedded devices. Here, the client-server architecture can be applied. Requests are initiated by clients and sent to servers. The servers process the requests and send any needed information back to the clients. Extra work must be done to ensure information is synchronized between different server instances, but the clients themselves can remain lightweight.

2.2.2 LDAP

The Lightweight Directory Access Protocol, or LDAP [8], is a directory service. A directory is a database optimized for reading, browsing and searching that usually contain attribute-based information. A directory usually puts more emphasis on filtering than on transaction or backup schemes found in traditional large databases, and directories are designed for high-volume search operations. Write operations, then, are a lower priority. LDAP is a popular choice for providing directory services, so an ambient computing system which incorporates it will be easier to integrate into an enterprise-level environment.

2.2.3 Speech Recognition

As computers become more integrated into our daily lives, interfaces for how people interact with them will change. One of the more natural interfaces is voice control. Speech recognizers usually use either a dynamic command vocabulary or a grammar-based vocabulary. Grammar-based vocabularies predefine a mapping of certain phrases to commands, at the expense of having to load a grammar. However, this allows for vocabularies that are more flexible and more complex than dynamic command vocabularies. Dynamic command vocabularies support a 'what-you-say-is-what-you-get' model in that all words are allowed, but only a few tokens in the correct order will trigger actions by the recognizer.

Examples of speech recognition software include Carnegie Mellon's Sphinx [28] and IBM's ViaVoice [14] software. ViaVoice is used for this system as it has both the above abilities, good overall recognition accuracy, and a Java API [15] to control it.

2.2.4 IEEE 802.11b

Wireless technologies have become inexpensive enough in the last two years to become a viable solution for connecting home automation devices together. Devices implementing the IEEE 802.11b wireless protocol in particular have been dropping in price and have become very popular. The 802.11b standard operates in the 2.4 gigahertz range, and allows for a maximum bandwidth of 11 megabits per second at a range of 160 meters. However, the maximum range comes down in closed-air environments, such as offices or residential areas.

802.11b provides a physical layer link for devices, so that other protocols, most notably IP, can be used as the transport layer. IP support also allows easy network configuration using DHCP (Dynamic Host Configuration Protocol), supported by many operating systems today. Encryption of the wireless link is also supported, although at the time of this writing, WEP (Wireless Encryption Protocol), the main protocol used for the majority of this encryption, was recently discovered to be vulnerable. However, the impact of this can be minimized by using application-layer encryption such as SSL or suitable VPN technologies.

2.2.5 Bluetooth

Bluetooth is another wireless protocol useful for creating local groups of devices. Bluetooth is a short range, low bandwidth protocol in the 2.4 gigahertz range that seeks to handle the connection of devices within a range of 10 meters, leaving the connection of devices outside this range to IEEE 802.11b. Bluetooth was originally designed to act as a cable replacement, eliminating needs for cables between computing components such as keyboards. However, it can be used as a replacement for 802.11b in close ranges for devices with low power requirements such as personal digital assistants and mobile phones. It can be used as another link-layer protocol to establish a network controlled by the ambient computing system.

The Bluetooth specification has had enough backing by industry in the last two years to deserve consideration, but at this time, not enough consumer devices have appeared to spark large consumer interest in Bluetooth. IEEE 802.11b has come down in price as well, thereby competing with Bluetooth's market.

2.2.6 X10

X10 [30] is a proven twenty-year-old technology used to send control signals to devices over residential AC wiring. A command module sends commands to X10 modules plugged into power outlets using short signal bursts to transmit a few bits of information. A binary one is sent as a one millisecond burst of 120 kHz at the zero crossing point of the AC power line, and a binary zero is recognized as the absence of this burst. A X10 ”packet” consists of eleven bits of information sent over eleven cycles. Two bits represent the start of the packet, four bits represent the address of the destination device, and the last five bits are actual information.

Uses of X10 include home lighting control, temperature control, and home security, as well as recent enhancements such as audio-visual component control and video surveillance cameras. However, X10 does have problems, such as:

- problems when sending signals across phases in home wiring,
- a lack of acknowledgements, and
- packet transmission errors due to line noise.

The last two problems make for an unreliable protocol. Many real world applications have to overcome this by sending the same command repeatedly to insure reception, limiting the applications that can use X10.

2.2.7 HomeRF

HomeRF [13] is another wireless technology in the 2.4 GHz range that provides wireless home networking. HomeRF implements quality of service support and adaptive frequency hopping technology to combat problems of latency and interference seen with 802.11b and Bluetooth. At the time of this writing, most HomeRF products are geared toward wireless networking and audio-visual products, although the forum envisions wireless data tablets, HomeRF telephones, and control systems tying in lighting, temperature control and security, much like X10 home systems. However, 802.11b has been widely adopted, leaving the future of HomeRF in doubt.

2.2.8 SOAP

The Simple Object Access Protocol (SOAP) [2] is a XML protocol for exchanging information in a decentralized, distributed environment that aims for interoperability. SOAP defines a message envelope format for describing a message's content and how to process it, a set of encoding rules for application-defined data types, and a convention for representing remote procedure calls and responses. SOAP is derived from the XML-RPC standard, and is a key piece of Microsoft's .NET strategy.

2.2.9 Service Location Protocol

Service Location Protocol [12] is an IETF standards track protocol that provides a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks. It aims to aid administrative setup of services and provide a way for programs to "auto-discover" services. Applications are treated as clients that have to discover servers within the network they are querying. Once a server is found, the server checks its list of services, or consults a local database, to determine the network address of the desired services. It then sends this information back to the application. This ability to dynamically determine services in an area is important to an ambient computing system. The system's behavior in a given domain is determined by the services it can perform, and this behavior will affect a user's decisions and abilities while there.

2.2.10 Universal Plug and Play

Universal Plug and Play (UPnP) [5] is another service discovery protocol. It follows a distributed architecture using TCP/IP, UDP, HTTP, XML and SOAP. Non-IP protocols can be used when necessary, though. Devices first obtain an IP through DHCP or Auto-IP, then send multicast discovery messages to find the root device for that domain. They then exchange XML messages describing the services they provide and parameters for each command, or action. Service descriptions are based on a standard template created by the UPnP Forum and are written by the device vendor. Device descriptions can be retrieved by the root device by issuing a HTTP GET request to a location hosting the device description. UPnP services respond to actions, and control points can register for notification when this actions occur.

However, at this time the UPnP architecture does not integrate the infrastructure for personalization as tightly as the architecture described in the next chapter; in fact, no mention of personalization is made. Also, searches for devices are not matched to client instances, meaning that clients may receive results that do not match the original criteria of the search. It is unclear from the UPnP API whether this will require a new search if an incorrect result is received, or whether the client will receive multiple results that can be checked against the original search criteria. Either way, searches would seem to be a little inefficient.

2.3 Other Projects and Products

2.3.1 Home Automation Systems

Home automation systems and products do not provide the intelligence needed to interact with the user on any more than a cursory basis. Most systems in this category allow the user to control devices remotely, but do not have the ability to dynamically change behavior. Although these devices lack much logic, they can be used and easily integrated as devices into the ambient computing system.

2.3.2 HP Chai

Chai [4] is an integrated suite of software products, development tools, and services for designing information appliances. Chai software blocks provide certain base functionality that developers can piece together to provide a platform on which to start design of the appliance. Hewlett-Packard provides the architecture, but not the design. This allows any sort of appliance to be built. But, as with home automation systems, the infrastructure to provide the user with a meaningful interface and experience to a pervasive system is not present. These products, however, would be useful as devices in an ambient computing system, as development time for new hardware interfacing to such a system could be significantly reduced.

2.3.3 Ninja

The Ninja research project [10] has prototyped a software infrastructure to support next-generation Internet-based applications. The infrastructure of Ninja is built to support services that are scalable, highly available, and tolerant of faults. Persistent state of services is usually stored at scalable and reliable data stores known as bases. All other state can be regenerated if lost. Operators within the Ninja system determine how data is transformed when moving from one service to another. These operators determine the behavior of active proxies which transform information from the bases sent to end user devices such as personal digital assistants and cellular phones.

The Ninja project does seem to provide a sensible architecture for providing state and data to devices. It does not address user interaction, however, since it is primarily a service architecture.

2.3.4 MIT Project Oxygen

Like many pervasive computing projects, MIT's Project Oxygen [19] assumes that computational power and devices will be abundant as air (or oxygen) in the near future. Natural spoken and visual interfaces will be used to collaborate, access information, and automate certain tasks. The interfaces will be embedded and nomadic, and the infrastructure as a whole will always be active, even though components may be down.

Oxygen is split into four core technologies:

- Handy 21 (H21) – H21 is a custom, reprogrammable handheld device used by Oxygen users. It will combine the functions of many handheld devices, such as cameras, personal digital assistants, and cellular phones.
- Enviro21 (E21) – E21 is similar to H21 but stationary. It is the workstation in the Oxygen system, meant to service a single space, such as a room.
- Network 21 (N21) – N21 is a flexible, decentralized network. It configures collaborative networks, provides service discovery and secure communication, and adapts itself to changing network conditions.
- Speech and vision technologies – These will provide the user with a natural interface to the Oxygen system.

2.3.5 Microsoft .NET

.NET [6] is Microsoft's platform for XML Web services, allowing applications to share data and communicate regardless of operating system or programming language. Clients using Microsoft operating systems will use SOAP messages to directly talk with applications in other locations hosted on servers running .NET enabled Microsoft server applications. Microsoft has also developed Visual Studio .NET to allow developers to build, deploy, and run XML Web services. Microsoft's first .NET service set is Hailstorm, built around the Passport ″one login″ authentication system. With ″HailStorm″, users receive relevant information, as they need it, delivered to the devices they're using, and based on preferences they have established.

.NET does seem to have personalization and a user-centric feel very much in mind. However, very few details of the underlying architecture are easily available. However, with Universal Plug and Play and .NET combined, the resulting architecture will be one that provides useful features that can be taken advantage of with an ambient computing system.

2.3.6 Other Ambient Computing Systems

An ambient computing system [21] was implemented by Steve Pennington at the University of Kansas. This implementation used a server-client architecture in which devices are managed by the clients. Events generated by the devices were sent to the server, which sent back commands to run. A system of preference use and storage was present as well. The system implemented by Pennington was purely event driven, and devices registered to trigger events. However, this becomes cumbersome when a large amount of events have to be registered. For example, a user interface such as a web interface would have to register many events for every device it would control. By having the ability to directly send actions, as well as registering for events, interfaces become much simpler to create.

2.3.7 ACE

The Ambient Computational Environments (ACE) project [3] aims to provide universal access to computational resources in the business and research world. The project foresees users interacting with their immediate environment and long-lived, robust services. Services are separate into two parts: the daemon that provides the service, and a command interface to the service. Applications then use these services instead of interacting directly with the operating system. The ACE project plans to create such a system to study the impact of ambient computational environments on networking systems and to research the mechanisms required to secure content.

2.4 Our Approach

The architecture described in the next chapter aims to provide a personalizable, robust ambient computing system that realizes and tests the viability of a new model for pervasive computing. When possible, proven (or very soon to emerge), open standard technologies were used to ease integration with other pervasive computing projects.

Chapter 3

Architecture

3.1 Overview

In a pervasive computing environment, all computer-controlled devices in a user's environment are linked into one seamless network. Stand-alone appliances, such as audio-visual equipment, household appliances and environmental controls, can now interact to provide more useful services to users. The ambient computing system architecture described here provides a software based, flexible, easy to use computing and networking environment that is compatible with current and future devices.

3.1.1 Meta-Operating System

A meta-operating system is a system the coordinates a diverse set of computing, networking and software resources allowing them to be used more easily throughout home and business environments. An meta-operating system has properties of traditional operating systems, such as:

- input and output control,
- operations and job control,
- scheduling, and
- separation of mechanism and policy at the device level, much like the device driver model of Unix operating systems.

By applying the operating system paradigm to a distributed networked environment, higher-level services such as personalization, presence (localization of services) and permissions (authentication, authorization and accounting) can be offered to users, applications and devices in a natural way. A meta-operating system is not just a device or service discovery and startup mechanism, or a control channel, but rather a complete architecture to provide multi-user, multi-tasking operations throughout a distributed environment with smart devices.

3.2 Requirements

3.2.1 Preferences and Personalization

An ambient computing system must be just that: ambient. A user needs to personalize such a system to fit his or her lifestyle. Therefore, personalization must be built into the core of any such system. This can be accomplished by enabling a user to set preferences for how certain actions or events affect the system. This allows the user to determine the behavior of the system and tailor it so that it becomes an invisible part of his or her life. Most products in the markets previously described are not focusing on a user's experience with the system, but on the underlying network.

3.2.2 Presence

An ambient computing system must also be context-sensitive. Events and commands may perform different actions based on the current environment. A user has a presence: he or she occupies a given location, which affects how the ambient computing system interacts with the user. By using the user's location as another input, an ambient computing system can adjust its behavior accordingly. Applications will take advantage of understanding presence to provide unique experiences such as music following a user from room to room or lighting the way through a darkened house for a user with their hands full.

3.2.3 Permissions

Many users will use the meta-operating system, each with a different set of preferences. Therefore, an ambient computing system must protect the integrity of the system and privacy of each user. Although network-layer security may protect the user's session with the system, more fine-grain control is needed to protect the resources that the user utilizes during his or her interaction with the system. Permissions and access control lists will protect the user's preferences and data as well as control the usage of devices within the system. This also has the benefit of enabling levels of trust. The user can determine who can access their information and the system can restrict or permit access to devices.

3.2.4 Transport Technologies

Ethernet can be used as a standard transport layer, as most traffic in an ambient computing environment is delivered over TCP/IP. However, wireless technologies will be key to providing universal access. With 802.11b, Bluetooth and HomeRF, the wireless network options are quite diverse. The X10 protocol can also be used, as the meta-operating system can translate messages and data into commands to send to X10 modules, allowing it to handle home control networks. The meta-operating system must integrate with all these transport-layer technologies. This will also allow the consumer many choices for connection technologies.

Where applicable, the meta-operating system must incorporate application-layer encryption, authorization and authentication as these are not present in the transport technologies. Technologies in this area include SSL, VPN technologies, and Kerberos.

3.2.5 Interfaces

A XML-RPC or SOAP interface to an ambient computing system will allow other systems outside the ambient computing environment to be easily integrated. This is especially true with Microsoft's .NET initiative.

SLP can be used as the interface for connecting to services within an ambient domain. This will leave client side configuration to a minimum. An interface to services managed through Universal Plug and Play is also desirable.

The database interface for the meta-operating system will be abstracted out in order to use many different databases from the same API, much like Java's JNDI ^{TM}[25] and JDBC ^{TM}[26] APIs. User preferences and device information needs to be stored. Fast reads from the database are needed to decrease user interaction delay. Integration with Microsoft's Active Directory is also desirable. The LDAP directory service meets these requirements.

3.3 Features

The above requirements can be used with the background learned from other ambient computing projects to determine the basic features of the meta-operating system, which are as follows:

- The meta-operating system must first function on standard PC hardware, with the ability to work on embedded systems being a high priority as well.
- The meta-operating system must send messages using TCP/IP. This means the communication between the devices and the core must take place using sockets. This also allows any medium that can transport IP to be used.
- Messages must be in text form so interfaces and systems different from the meta-operating system will be able to easily communicate with it. XML messages will allow messages to be structured simply.
- Since there are a myriad of consumer devices, a common interface to all devices is required. Most device-specific logic will reside in the module which uses this interface to communicate with the system, where the core logic will reside.
- Devices and users will enter and leave the system at any time. All data structures, device capabilities and user profiles must be dynamic and learned.
- The system must be event responsive. Devices can wait for external events to occur, then trigger the appropriate actions to respond.
- The system must allow the user to customize most components of it. Therefore, the system must provide storage for information about users, devices and any other pertinent data.

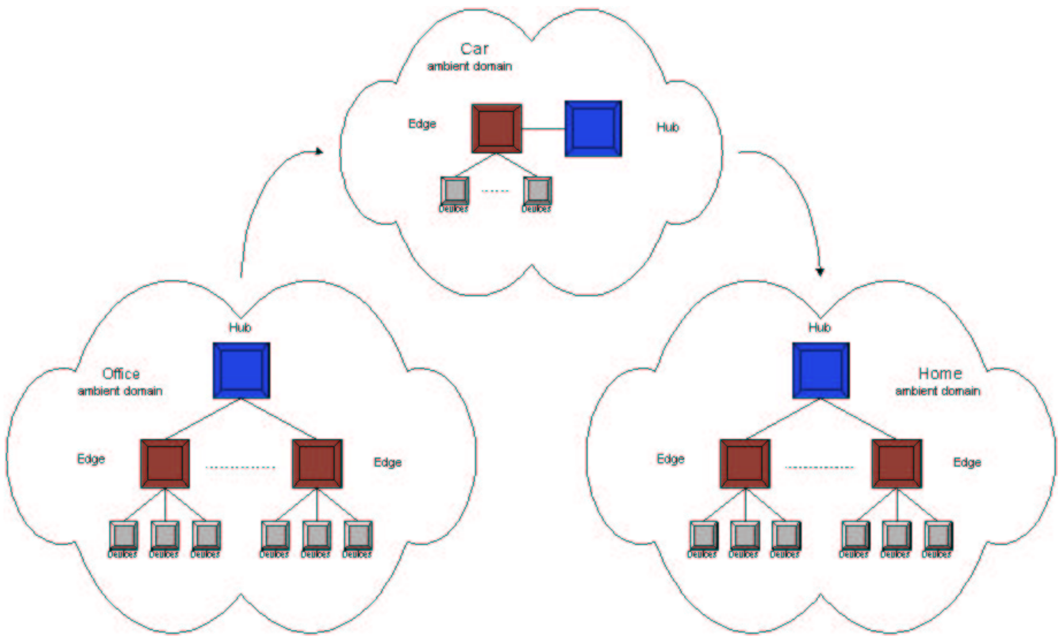

Figure 3.1: Logical Architecture Model

- The system must be able to identify and manage multiple users.
- The system must protect user information and conduct communication over secure channels.

3.4 Architecture Model

The architecture of the meta-operating system is most closely described by the client-server model. The server, or hub, is the ”kernel” of our meta-operating system, encapsulating the core logic and services much like a kernel in an operating system. The hub accepts client connections, provides initialization data for the clients, waits for messages from the clients, and executes operations on itself or the clients. The client, or edge, manages the sending of messages to and from the hub for each device connected to the meta-operating system for which it is responsible. Through the devices, user and device input is accepted and used to create messages to send to the hub. These messages perform operations on the hub, and information is sent back to the edge as necessary. The edge then sends this information to the devices. The devices perform device-specific actions based on the information sent to them. A ambient domain is a collection of edges controlled by one hub and maintained by one administrative entity, such as a business or homeowner.

An implementation based on this architecture is described later in the document. This chapter details the following areas of the meta-operating system:

- Device Management and Addressing
- Hub-Device Communication Protocol
- Inter-domain Communication
- User Management
- Preference Management

3.5 Device Management and Addressing

3.5.1 Device Model

Devices in the meta-operating system represent the division between the controlling software of the system and the hardware used to receive stimuli and send responses to the physical environment. Devices can be responsible for any number of physical inputs. Inputs from the environment, such as a light being turned off, an input to a CGI script from a Web browser, or motion from a motion detector are mapped into events or actions to send into the meta-operating system. Responses then come back from the system towards the devices. The device then receives its action and takes the appropriate steps to complete, such as turning a light on in case of motion, displaying dynamic content based on the input to the CGI script, logging movement into a database, or sending a message to the device in the device's own protocol.

3.5.2 Edge

The edges act as connections between the devices and hub in an ambient domain. When created, the edge initializes itself and any devices that it manages. The edge then creates a connection between itself and a hub, allowing edges and devices to be distributed on different hosts than the hub. Messages sent from devices route through the edge to the hub, and messages from the hub pass through the edge first so the edge can send the message to the specified device. These messages contain event and requests for actions or information. Once messages are delivered back to devices, the devices have the information necessary to perform further actions or events. Edges can initiate requests as well.

3.5.3 Hub

The hub is the server in the server-client analogy and is the core service point for the meta-operating system. All messages, whether between device and hub or between hub and hub, are routed by the hub. It controls every part of the ambient system, and is responsible for the following tasks:

- Receiving messages from devices as routed by the edges connected to it.
- Receiving messages from other hubs.

- Controlling permissions on devices, events, and actions to ensure their requested execution is allowed.
- Maintaining the state of all devices in its domain.
- Managing a connection to the database used by the ambient system.
- Performing events and actions according to the requests sent by devices.
- Sending messages to devices and other hubs in response.

3.5.4 Addressing

Addresses in the meta-operating system consist of a globally unique name for each device, edge, or hub. Devices in a domain have addresses that are unique within that particular domain. Addresses are hierarchical, i.e., a device's address represents the routing levels that a message must go through in order to reach a given destination. Devices can be assigned addresses based on configuration files, or in some cases, dynamic generation.

For example, an address of the form

```
hub0.edge0.client0
```

shows that messages from the device 'client0' will travel first to the edge 'edge0', which will route them to the hub 'hub0'. So, in the example above, the edge sends all messages prefixed with 'hub0' to the connection with the hub it created. When a edge first connects to a hub, the hub does the same, creating a mapping between the new connection and the name of the device sending the message (in this case, the edge name 'edge0'). Since routing is name-based, descriptive names such as

```
NicholsHall.Room220.edge0.mp3player0
```

can be used much like DNS entries: to map address into easily read names.

Each domain of addresses is managed by one hub (or more if replication of services is desired). Each domain will have a public, globally unique address. This can lead to globally addressable devices such as:

```
mydomain.myhouse.livingRoom.overheadLight
```

3.6 Hub-Device Communication Protocol

The meta-operating system uses XML-formatted messages to send information. Messages are sent as text over network streams, making it easier for other systems to read and therefore communicate with the meta-operating system. The system uses the following message types:

- Registration messages
- Event messages
- Action messages
- Information messages
- Miscellaneous messages

3.6.1 Registration Messages

Registration messages are used to create entries for devices, events, and actions. These registration messages are always sent to the hub, so that by registering, the devices can initiate and respond to events. Once a registration message is sent, a message denoting success or failure is sent back to the device; this message is described in more detail below.

There are five types of registration messages:

- Registering a device. A given name is sent, and this name is used as the address in the meta-operating system for communication with other devices.
- Registering an event. Given a device name and an event name, this adds an entry on the hub stating that the given device can generate the given event.
- Registering an action. Once this message is complete, the given device can perform, or trigger, the given action.
- Registering a ”todo” item. This message is used to build lists of actions to take when a given event takes place. This is how command macros are built up.
- Registering an edge. This message is used by the edge to send addressing information to the hub when the edge’s connection to the hub is first created.

3.6.2 Event Messages

The event message is used by a device to tell the hub that an event has occurred. Devices as commanded by the hub then execute the correct actions that this event triggers. Data sent along with the event message is used by the hub for arguments to the actions it executes. The hub sends a message back to the device denoting whether the event was successfully received.

3.6.3 Action Messages

The action message is sent from a device or the hub. It tells a given device to execute a command, whose implementation is device-specific. Data is sent with the action message and used for arguments to the command.

3.6.4 Information Messages

There are two types of informational messages in the meta-operating system: messages which utilize the database, and messages which utilize the device tree data structure.

The first type of informational messages are sent by devices to query for or modify data in the database connected to the hub. With this interface, devices can use the database without having to open a connection to the database or without having to know what type of database it is. Both users and devices have data associated with them that can be queried, added, modified, or deleted as events or actions take place.

The second type of informational message is used to query the device tree data structure to discover what devices and actions are registered with the hub.

3.6.5 Miscellaneous Messages

Success and failure messages are sent in response to most messages. The success message indicates the previous operation was completed successfully. The failure message does the opposite, but also includes data such as error codes and messages to tell the user what went wrong.

3.7 Inter-domain Communication

Communication across ambient domains allows user and device information to be exchanged across public networks, allowing coordination between a user's workplace, home, and using wireless technologies, his or her personal space as well. A secure protocol and mechanism for accessing devices and information from different domains using strong encryption and authentication is necessary.

3.8 User Management

The personalization of the meta-operating system for use by a user must first identify the person using the system. Identification takes place at an input point of the system. This can be a Web login, speech spoken to a voice recognition system, use of a hardware token such as the iButton ® [7] or a smart card, or other biometrics such as fingerprint readers. Once the given credentials are presented by the user, the meta-operating system can validate the user against the appropriate credentials in its databases, and generates a token for the user to use during his or her session.

Each user has their own profile as well, stored in the domain's database. This consists of personal information, such as their:

- user name,
- groups they belong to,
- password,
- full name,
- address,

- telephone number,
- email address,
- picture,
- current location,
- preferences,

and others. With the correct permissions and authentication methods, this profile can be accessible from multiple domains, allowing a user to access his or her profile from other locations. By tying together the user and domain name, pieces from a user's different profiles can be used in several different domains, leading to user addresses of the form:

user@mydomain.com

3.9 Preference Management

Preferences are at the core of the meta-operating system. They are the basis for what the user will ultimately see and how the user will interact with the system; therefore, their management is an important part of the ambient system. The meta-operating system comes with a basic set of preferences to present the user with a standard behavior. Personalization takes place through the use of wizards to configure parameters and arguments used by devices when executing events and actions. The preferences can then be saved in a database for future and remote use.

Preferences fall into two categories: user preferences and device preferences.

3.9.1 User Preferences

User preferences are part of the user′s profile. Events and actions use user preferences as arguments, allowing the user to personalize how an event or action affects the ambient system. For example, a command ″Play rock music″ spoken to a voice recognizer will cause the system to play different songs dependent on the person and his or her preferred rock song listed in their preference for this command. Menus displayed to the user can be modified based on a user′s preferences and the output device the user is currently using.

Command macros can be created by the user to chain together a set of commands into one usable command. For example, a user could create the command ″Relaxed mode″ which would:

- play soothing music
- dim the lights by 30%, and
- disable the doorbell and telephone for thirty minutes.

A set of stock macros will be installed by the meta-operating system and configured by the user to suit his or her tastes.

3.9.2 Device Preferences

Device preferences are stored as part of the device's profile, much like a user profile. Device preferences are used to control the default behavior of a device. These defaults can be changed by a user, but these changes will be saved in the user′s profile, not the device's. Precedence can be set by the device to base behavior on a user′s preferences first, or whether to use the device's preferences by default.

3.10 Permissions

Each device, event and action in the meta-operating system has an associated identity and access control list. This is modeled after the Unix style of permissions, where each file has a user, group, and permission bits set. The messages sent in the system specify operations to perform. The messages also contain identity information, so the system can check whether the device, event or action allows this operation, given this information. Extended access control lists for granting or denying access to objects based on any criteria will provide finer-grain control as well.

Chapter 4

Implementation

4.1 Overview

This chapter details an implementation, referred to as the MetaOS $^{\mathrm{TM}*}$ that meets the requirements set forth in the architecture chapter. This implementation was written using the Java $^{\mathrm{TM}}$ 2 platform, standard edition [24] so that both Unix and Windows hosts could be used for development and execution. Figure 4.1 shows the physical layout of a MetaOS system. Each section is described in more detail below.

4.2 Hub

The hub, as stated before, is the ″kernel″ of the MetaOS. Through edges, all devices are connected to a hub. The hub receives messages, performs operations based on these messages, modifies the database if necessary, and sends messages to the correct destination.

*MetaOS is a trademark of Ambient Computing, Inc.

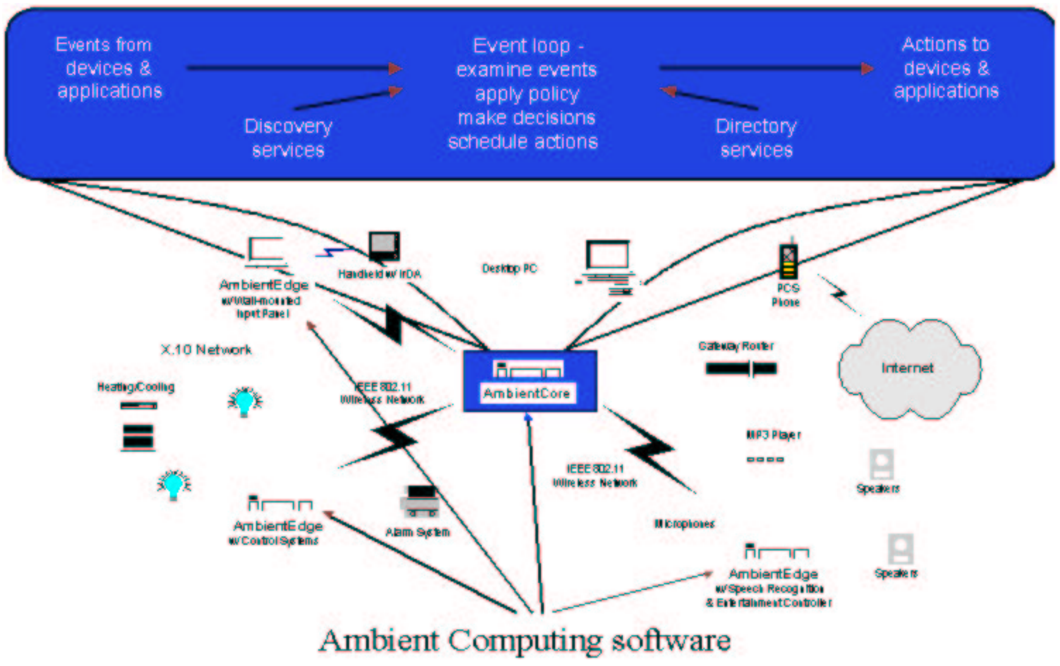
Events from devices & applications
Discovery services
Event loop - examine events apply policy make decisions schedule actions
Directory services
Actions to devices & applications
AmbientEdge w/Wall-mounted Input Panel
Handheld w/ IrDA
Desktop PC
PCS Phone
X.10 Network
Heating/Cooling
IEEE 802.11 Wireless Network
AmbientCore
Gateway Router
Internet
MP3 Player
Speakers
IEEE 802.11 Wireless Network
Alarm System
Microphones
AmbientEdge w/Control Systems
AmbientEdge w/Speech Recognition & Entertainment Controller
Speakers
Ambient Computing software

Figure 4.1: MetaOS System Diagram

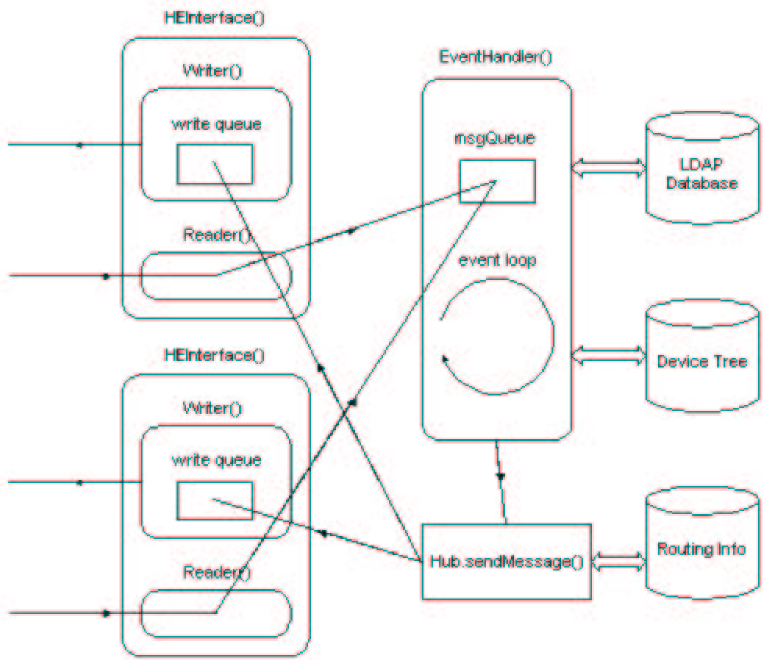

Figure 4.2: MetaOS Hub

4.2.1 Initialization

The hub process, shown in Figure 4.2, is executed with one argument, the location of the configuration file. This file is an XML document describing values for:

- Hub port – what IP port on which the hub should listen for incoming connections.
- Encryption type – the encryption type of the listen socket the hub creates.
- Device prefix – the name of the hub in this MetaOS domain. This is the root prefix of all device names in this domain.
- Log filename – the name of the log file for this hub, where debugging and other informative messages are sent.

The next step is to start a thread to handle the processing of messages that the hub receives. The event handler thread is discussed in the next section.

The hub then performs the normal function of most servers: waiting for connections, then performing operations based on messages sent to it. The hub knows the port for the listen socket from the configuration file, so it creates a listen socket. Once the listen socket is created, the hub enters a loop, accepting connections until the hub is terminated. The hub thread blocks until a connection is made on the listen socket by an incoming edge connection. The hub then passes this new connection's socket address as an argument to construct a network interface thread, which is described in further detail below.

Once this thread has been created, a stream exists between the hub and edge. Messages are sent and received over this stream. The first message sent by the edge contains routing information for this edge. The hub takes this routing information and

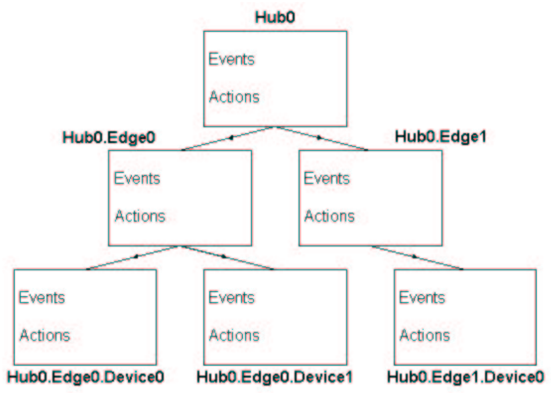

Figure 4.3: MetaOS Device Tree

adds it to a table, mapping the connection to the routing information. After this is complete, the hub continues waiting for connections from other edges.

4.2.2 Event Handler

The event handler thread controls most of the logic in the hub. The event handler's job is to process events as they arrive into the event queue from the edge connections.

Initialization of the event handler thread first creates a tree data structure that contains information about all devices, events, and actions in the domain administered by this hub. This tree, as shown in Figure 4.3, is populated by the event handler thread in response to registration messages sent from devices. Once the tree data structure is created, a connection to the database is created for use in handling queries for user and preference information.

The event handler thread then waits until a message appears in the event queue. The thread determines what type of message it is, such as an event, action, or request for information. The different types of messages are described in further detail later in this chapter. Using the incoming message, the event handler thread traverses the device tree data structure, reading information from it or modifying it if permissions allow. If messages are to be sent back to the message's source, the message is sent to the correct edge interface thread, using the routing information stored in the routing table on the hub.

4.2.3 Edge Connections

The connection between an edge and a hub in the MetaOS is a bidirectional stream. Java methods are used to read and send text characters on the stream. Sending text messages allows non-object based implementations to easily communicate with the hub.

The following message transmission architecture relieves blocking, requiring one thread on the edge, and one thread per edge on the hub. Blocking can first occur when reading or writing an object to the stream. Program execution stops at a read or write to the stream until it finishes. We can use a queue to store these messages until they can be read or written, using a separate thread to block on reading and writing objects to the correct stream. Program execution can then proceed in the main thread, with the only delay being the retrieval from or placement into the queue.

Blocking can also occur when the edge first connects to the hub. In C, the select() method can be used to poll a socket for a connection. If no connection is present, control returns to the calling program. In Java, the accept() method blocks accepting a connection on a socket, as in C. However, Java does not have a select() method. Therefore, to minimize blocking on accept(), one thread must only accept connections. However, the hub's only real job after initialization is to accept connections from edges. Since most of the execution takes place in the event handler thread, the hub can block

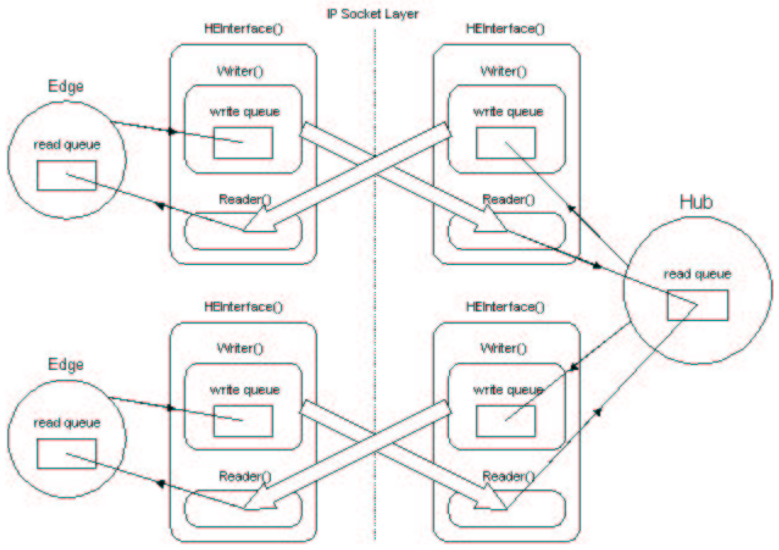

Figure 4.4: MetaOS Message Transport Architecture

on the accept() call.

The core of the message transmission system, as picture in Figure 4.4, is the network interface thread. It can be instantiated in two ways. Both versions of the thread take a queue as one argument. This queue is used to store messages that have been read from the socket stream to which the network interface thread attaches. The first version of the network interface thread takes as arguments a queue as described above and a socket. The hub uses this version to create a thread for each socket returned from the accept() method of the listen socket the hub created at initialization. The second version of the network interface thread takes as arguments a queue as described above and an IP address and port pair. This version is used by the edge to create a socket to connect to the IP address and port of the hub's listen socket.

Once either version of the thread receives the correct arguments, receive and transmit streams are created from the socket. A queue for holding messages to be transmitted is created, mirroring the receive queue the thread has as a parameter. Two helper threads are then created. The first helper thread is a reader thread. The reader thread starts, then blocks on reading text from the receive stream. Once a complete message is read from the stream, the reader thread places the message in the receive queue passed to the network interface thread as an argument, then blocks to wait for the next series of text characters. The second helper thread is a writer thread. It blocks until an message is placed in the transmit queue by the used of the send() method of the network interface thread. It then writes the text of the message to the transmit stream and blocks until another message is placed in the transmit queue.

The edge or the hub then use the send() method of the network interface thread to send messages, or block reading messages from the receive queue. The edge then processes these messages and blocks again, while the event handler thread on the hub reads the messages on the hub, processes them, and continues waiting. The first message sent by an edge contains routing information. The hub side of the network interface thread takes this information and maps the routing information to that thread in the hub's routing table. When messages from the hub are sent to an edge, the routing table is used to select the correct thread to send the message on, thereby ensuring the correct edge gets the message.

4.3 Messages

The original reason XML was chosen as the MetaOS message format was the ability to use XML to transform one set of data into multiple formats based on the output device presented to the user. However, XML is an emerging standard with many toolkits available. Its structure is more descriptive than normal text, and can be constrained easier than other message formats. All these reasons lead to the use of XML in MetaOS messages.

This version of the MetaOS had the following messages types, described below:

- Registration messages
- Event messages
- Action messages
- Informational messages
- Miscellaneous messages

Almost all messages have two elements in common: the <identity> and <acl> elements. The <identity> element contains user and group information. The <acl> element contains a string representing a Unix permissions bit mask. Each object in the MetaOS has a corresponding owner, group owner and ACL. These are used, as in Unix, to determine whether the operation the message represents has permission to access the necessary resources. This check is performed in the event handler thread before any operation is allowed to complete.

An example of an XML message is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322259806">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.316394043</message_ID>
  <register>
   <identity>
      <user>lsanders</user>
      <group>adm</group>
</identity>
<device>
```

```
  <name>hub0.edge0.SOAPServer.client1</name>
  <ACL>111111111</ACL>
</device>
   </register>
</AmbComp:ambient_message>
```

Please consult Appendix B and Chapter 5 for examples and usage.

4.3.1 Registration Messages

The *register_device* message is sent to register a device into the MetaOS domain controlled by the device of the destination address. The identity of the sender is checked by use of the <identity> element. If permissions allow the creation of this device, a node is created in the device tree data structure managed by the event handler thread, with the given name (for example, hub0.edge0.SOAPServer.client1) and the given access control list settings (here, 111111111, which is used as a bit mask much like the file attribute bits in Unix). A reply indicating success or failure is returned to the address specified in the <source_address> element.

The *register_event* message adds an event to the event list in the correct device node in the event handler thread's device tree structure. The name is given in the <name> element. The device is the complete prefix of this name, i.e. all sections of the name before the last period. Therefore, event names must be described without periods. Permissions are checked using the <identity> element. If allowed, the event is added to the list of events this device can generate. A reply indicating success or failure is returned to the address specified in the <source_address> element.

The *register_action* message adds an action to the action list in the correct device node in the device tree in the event handler thread. The name is given in the <name> element; the device this action is added to is determined the same way as in the *register_event* message. If permissions allow after checking the <identity> element, the action is added to the list of actions this device can perform, or trigger, and a reply indicating success or failure is sent back to the originator of the message.

While the *register_action* message creates a single action to trigger, the *todolistitem* message creates a macro, or list of actions, to perform when a specific event takes place. As one may recall, an event is issued by a MetaOS device in response to external or internal stimuli. This event triggers a single action in the MetaOS system, which in turn may cause other events to happen. An event with a todo list simply triggers multiple actions. If permissions allow the todo list item to be added, the action is added to the todo list of the event specified in the <parent_event> element. The <name> element specifies the name of the action, while the <data> element provides arguments for the action. A reply indicating success or failure is sent back to the originator of the message.

The *HEInit* message registers routing information with the hub. The <name> element contains the root prefix of all messages that the registering edge should be sent. This name is then mapped to the thread that the message was seen on. When sending a message, the hub checks the root prefix of the destination address of the message. If the prefix is found, the hub calls the send() method of that thread, and the message is sent. If it is not, the message is logged, and then dropped.

4.3.2 Event Messages

The *event* message is sent by a device to signal the devices specified in the <destination_address> element to take action based on the event that just occurred. This device should be a hub, since it is the only device that will have access to the device tree data structure. If permissions allow, the hub executes the actions in the todo list of the event named in the <name> element with the optional data given. A reply indicating success or failure is sent back to the source of the message.

4.3.3 Action Messages

When the hub receives an *action* message, the hub instructs the correct device to execute the action it has registered to execute named by the <name> element with the data specified in the <data> element. The device name is specified in the <destination_address> element. The device specified in the <source_address> element is sent a reply indicating the success or failure of the action's execution.

4.3.4 Informational Messages

In each of the informational messages, the type attribute of the <name> element determines what type of database entry on which the message operates. This allows the MetaOS to change the behavior of the operation. Currently, this is used to switch the mapping of the <name> element to a directory entry based on whether a user or device entry is being accessed.

The *query* message type asks the hub to perform a search on the entity specified in the <name> element and in the database section specified by the type attribute. The <data> element specifies arguments to use for the search. The arguments are attribute and value pairs. The attributes listed should have their values returned through the use of the response message type described below. The *query* message type is one of the few messages types that does not return an message indicating success or failure; the search results are returned instead.

The *response* message type returns the result of the database search specified by the *query* message type. The <message_ID> element in the <response> element specifies the message ID of the *query* message this message is answering. Each <result>

element specifies the database entity in the name attribute, and the values of the requested attributes in the <param> elements. Multiple results can be sent back to the query's source.

The *add* message type adds an value to an attribute of the specified database entry. The <name> element specifies the database entry, and the <param> element specifies the attribute's new type. Depending on the underlying database and schema, this may overwrite the previous value of the attribute, or just add another value to it. A reply indicating success or failure is sent back to the device specified in the <destination_address> element.

The *delete* message type deletes values or attributes from the database entry specified in the <name> element. The <param> elements specify attribute name and value pairs. The type attribute of the <delete> element specifies which type of deletion is to take place. A value of "value" tells the database to delete only the specified value from the list of values of the given attribute. A value of "attribute" deletes all values of the given attribute. Upon completion, a message indicating success or failure is sent to the destination device.

The *modify* message type changes the value of an attribute of the specified database entry. The first <param> element specifies the attribute value to be changed. The second <param> element gives the new value of the attribute. Only two <param> elements should be present in the <data> section; all others are ignored. A message indicating success or failure is sent after the modify operation is completed.

The *list* message type allows a device to query the hub for the current state of the device tree data structure in the event handler thread. The <listType> element specifies the type of information to be returned. If the <listType> element has a value of "device", the names of the currently registered devices located in the device tree with a root prefix of the name specified by the <root> element will be returned. Each device is returned in a <device> element.

If the <listType> element has a value of "actions", the actions registered for the device specified in the <root> element are returned. Each action is specified in an <action> element.

4.3.5 Miscellaneous Messages

The *ack* message type reports the success or failure of the previous operation issued by the device named in the <destination_address> element. The <message_ID> element in the <ack> element specifies the message ID of the message this *ack* message is responding to. The value of the <status> element is either "success" or "failure". If the *ack* message indicates failure, the <data> element can be used to return back error codes or messages.

The *deviceID* message is only used with the SOAPServer device. Remote clients communicating with the MetaOS through XML-RPC or SOAP-style calls can issue register, event, and informational messages and receive the correct replies using the normal server-client model of most web servers. However, the socket used to communicate with the client will be closed after this transaction completes, as is normal for HTTP communication. However, to send an *action* message to a client, the hub must be given a port to send the message to.

When registering to receive actions, a remote client issues a *deviceID* message. The port is specified in the <param> element, as the IP address of the client can be determined when it sends the message to the SOAPServer. The SOAPServer dynamically creates a MetaOS address that the remote client can use to register as a device. The

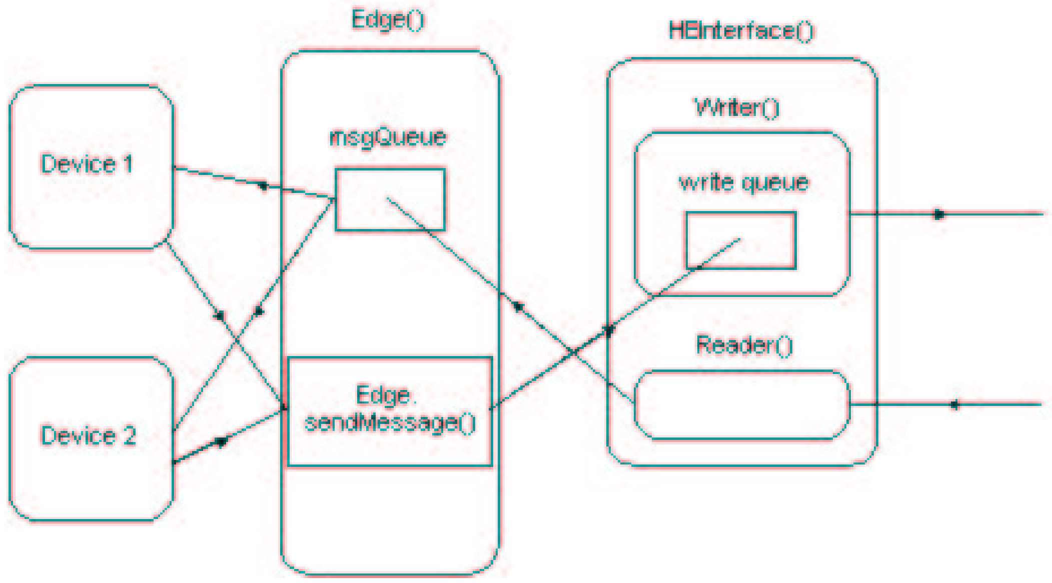

Figure 4.5: MetaOS Edge

SOAPServer maps this new address to the given port, and sends a response to the remote client containing this address in a *deviceID* message. The remote client can issue messages and participate as a MetaOS device. Messages sent asynchronously to this device first reach the SOAPServer device, which then consults its table to translate the given destination address to an IP address and port. The message is then delivered to the remote client using the same XML-RPC or SOAP-style calls.

4.4 Edge

The ambient edges act as conduits for messages between the devices and the hub in an MetaOS domain. The edge initializes a connection to the hub, initializes devices, and routes messages between the devices and the hub.

4.4.1 Initialization

Like the hub, the edge process shown in Figure 4.5 is executed with one argument, the location of the configuration file. The configuration file specifies the values for:

- Hub IP address – the IP address of the host on which the hub resides.
- Hub IP port – the port on the above host where the hub process has created a listen socket for accepting connections.
- Encryption type – the encryption type of the socket that the edge will create.
- Hub name – the name of the hub to which this edge connects.
- Device prefix - the name of the edge in this MetaOS domain. This is also the root prefix of all devices controlled by this edge.

The edge's configuration file also contains a list of devices to instantiate. Each device entry has two arguments: the name of the device to use as an address in the MetaOS domain, and the type of device this device will be. The second argument is used to determine which type of device object to instantiate.

Once the configuration file is parsed, the edge creates a queue for receiving messages from the hub. An object which provides random message IDs is also created. This is used to generate a unique ID for each message the edge sends to the hub. A network interface thread is created with the typical edge arguments as described in the edge connections section: an IP address and port number, and the receive queue created above. Once this thread is created, a *HEInit* message is sent to the hub specifying the edge name as read from the configuration file. Once the *ack* message is received, the edge registers itself in the hub's device tree data structure.

The edge then creates the devices listed in its configuration file. For every device in the configuration file, an object of the class specified is instantiated with the name given in the file. Together with the edge's device prefix, this name is a valid address in the MetaOS domain. Devices use the send() method of the edge's network interface thread to send messages to the hub.

Once all initialization is complete, the edge simply enters into a loop, checking the receive queue and blocking until a message from the hub is received. Once the message is received from the queue, the edge passes the message to the correct device (as specified by the <destination_address> of the message) for processing, and continues blocking for new messages.

4.5 Devices

Devices provide the separation of mechanism and policy in the MetaOS. A mechanism exists to send and receive messages to send events, actions, and information queries and operations. These are the ”system calls” of the MetaOS. This is separate from the implementation of the detection of stimuli and performing of commands to affect the environment, which is device-specific. Security can be enforced by requiring devices to authenticate with the MetaOS.

4.5.1 Device Interface

Each type of device is implemented in a class derived from the DeviceHandler base class. The base class takes the name of the device as its only argument. This name is the suffix of the MetaOS address for this device. The main methods of the DeviceHandler class are:

- getID() – returns the name of this device.

- getPrettyName() – returns the name of this device to use in menus and the like.
- setPrettyName() – sets the user interface friendly name of the device.
- sendMessage() – send the specified message to the edge managing this device.
- processAction() – receive a message from the hub (via the edge) and process it according to the logic specific for the device's task.

In addition, this class also defines helper functions to register events, actions, and todo list items, send *event*, *action*, and *ack* messages, and receive *list* messages from the hub.

The DeviceHandler class inherits the properties of a thread through the Java Thread class. Therefore, the base classes must define a run() method, which Java calls to start the thread on its instantiation. Within this method, the devices must register the events and actions they are responsible for. Then these classes wait for input, send messages to other devices and the hub, and wait for replies.

The following subsections provide some examples of devices created for use in the MetaOS.

4.5.2 X10Device and X10Monitor

The X10Device device is used to send X10 commands to X10 units via an external command. When instantiated, the device reads in its configuration file, which specifies the name of the external program, any arguments to it and a list of mappings. These mappings map the MetaOS name of the device to an X10 house code, which specifies the address of the X10 module connected to the external device to be controlled. The device then registers actions corresponding to each X10 module.

When one of the registered X10 actions is executed, the X10Device parses the *action* message sent it for any arguments to pass to the external command. For example, the argument can be the word ”on”, signifying that the X10 module should supply power to the external device connected to it. The external command is then executed with the correct house code and data, and an acknowledgement message is sent.

The X10Monitor device is used to send registered X10 events in the MetaOS system. Like the X10Device device, the X10Monitor device reads in its configuration file, determining the external program, arguments to it, and a MetaOS device name to X10 house code mapping. It also registers one event, ”X10Event”.

When started, the X10Monitor device executes the external command in monitor mode, and parses the output file generated by it. When an external X10 module generates an event, it is written in the output file. The X10Monitor device checks to see if the house code of the module is present in the mapping from the configuration file. If it is, an event of ”X10Event” is sent into the MetaOS system. The event’s data includes the house code and the name of the function that was completed by the X10 module (for example, ”off”).

4.5.3 FrameGrabber

One device in this version of the MetaOS is a device to grab frames from a TV capture card. The FrameGrabber device uses the Video4Linux [1] library, which interfaces with a C library used to control the TV capture card. The frames are grabbed when an X10 camera notices movement and sends X10 events. Another device registers as a X10 monitor. When the FrameGrabber device is instantiated and started, it registers one action, ”grabFrames”. It registers one todo list item for the X10 event registered by the X10 monitor, so that the ”grabFrames” action is sent to the FrameGrabber device.

The FrameGrabber device waits for an action of ”grabFrames”. This action requires two arguments passed to this device in the <data> section of the *action* message. The first argument is the number of frames to grab; the second argument is the directory name to which the frames are written. Once the arguments are read, the frames are read from the TV capture card and saved to the specified directory with a .jpg extension.

4.5.4 VoicePrefs

The VoicePrefs device works in conjunction with the voice recognition device that allows the user to map verbal commands to MetaOS actions to perform.

The VoicePrefs and VoiceRecognizer device also provide a good example of the separation of mechanism and policy in the MetaOS. The voice recognition device provides the mechanism. It uses IBM's Speech for Java [15] to take microphone input and generate the words as text, but only acts as a translator. It registers for one event, "spokenText". It then waits for an action of "addGrammar". Upon this, the dynamic grammar is loaded. The voice recognition device then waits for any input, parses the spoken words into text using the Speech for Java libraries, and issues an event of "spokenText" with the words as arguments. The voice recognizer device only provides a way to access the spoken words, in essence, providing access to the low-level medium of sound.

The logic and policy decisions are based in the VoicePrefs device. When started, this device sends an action of "addGrammar" to the voice recognizer device to load its grammar. It then registers to handle one action, "matchAction". The VoicePrefs device then reads in a voice configuration file, which specifies a mapping between a given set of words and an action with associated data. For example, the following example sends the action hub0.essence_edge.X10.light0 to turn the light off when the words "lights out" are spoken.

```
<preference>
    <spokentext>lights out</spokentext>
    <action>
        <name>hub0.essence_edge.X10.light0</name>
        <data>
            <arg>off</arg>
        </data>
    </action>
</preference>
```

The voice preferences device registers one action, ”matchAction” under the ”spokenText” event of the voice recognizer device. When this action is received, the voice preferences device matches the given words, and executes the specified action from the voice configuration file with the given arguments.

4.5.5 Mp3Player

The mp3 player device plays mp3 audio files and streams, outputting the sound using the sound card of the host it resides on. The mp3 device registers actions for common audio control operations, such as pause, play, volume changes, and many others. The most common argument, if needed, is the name or location of the mp3 audio file or stream to play. The device then waits for actions. Upon receiving them, it translates the given action into the correct operation on the mp3 player software. The player application is determined by the device's default preferences or by user-defined preferences if defined.

4.6 Database Connections

The database holds all user, device and content information. Java APIs are provided for interfacing to directory services. The Java Naming and Directory Interface TM

(JNDI) [25] is a standard extension to the Java platform, providing Java technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise.

4.7 LDAP

LDAP was chosen as the directory service for the MetaOS, as our architecture design needed a fast, lightweight database (see Section 3.2.5). Writing and transactions are not as important as fast reads from the directory. In keeping with the use of open source packages, the LDAP server used for this implementation of the MetaOS is the OpenLDAP LDAP server version 2.0.11 [9], a free open source implementation of LDAP version 3 [29].

The LDAP directory structure for the MetaOS is divided into two sections: one for users, and one for devices. Each entry in the LDAP directory is specified, sorted, and stored by its distinguished name, or DN. The DN specifies all name sub-components necessary to traverse the directory to the specified entry. Following the advice of RFC 2247 [17], the major divisions of the directory, users and devices, are specified using the domain component, or dc, attribute. If the DN is composed of dc attributes, the dc attributes will specify the host name of the LDAP server to contact. User names are mail IDs, which are globally unique by default [11]. Device IDs will be the MetaOS address they use. This structure is illustrated in Figure 4.6.

The LDAP directory is constrained by a schema specifying which attributes a directory entry can have. LDAP entries can inherit attribute definitions from other object class definitions. The success or failure of a LDAP operation is conditional on the operation acting within the boundaries of the schema definition. The schema used by the

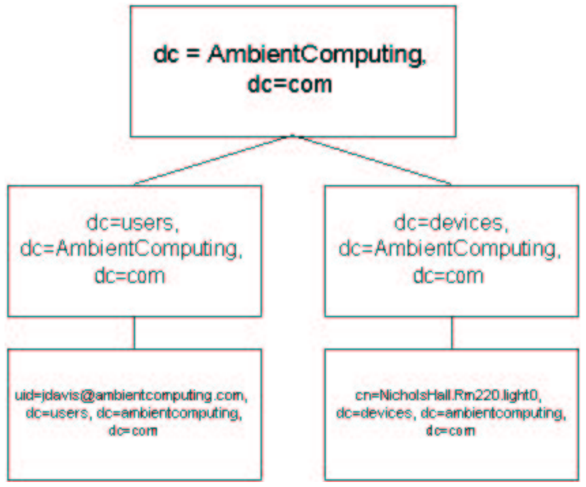
dc = AmbientComputing,
dc=com
dc=users,
dc=AmbientComputing,
dc=com
dc=devices,
dc=AmbientComputing,
dc=com
uid=jdavis@ambientcomputing.com,
dc=users, dc=ambientcomputing,
dc=com
cn=NicholsHall.Rm220.light0,
dc=devices, dc=ambientcomputing,
dc=com

Figure 4.6: MetaOS LDAP Directory Structure

MetaOS is defined in Appendix C.

4.7.1 DirLookup.class

The DirLookup class encapsulates the necessary JNDI methods for interaction with the LDAP directory. When creating an object from this class, the only argument required is the location of the LDAP configuration file. This file lists:

- The Java class specifying the interface to use for accessing this specific type of directory. This is known in Java as a factory.
- The LDAP URL for accessing the root of the directory.
- The type of authentication to use. This can be password-based, or SASL with SSL can be used.
- The security principal. This is used as the identity token for authentication.
- The password for validating the security principal.

After the connection to the directory is created, the following methods are then used to access and modify the directory entry argument:

- getValue() – This returns back all values of the given attribute of this directory entry.
- addValue() – This method adds the given value to the given attribute. If allowed by the schema definition, this may give the attribute more than one value. If the attribute already has a value, an error is thrown.
- replaceValue() – This method replaces all values of the given attribute with the given value.
- removeValue() – This removes the given value from the set of values for the given attribute.
- removeAttribute() – This removes all values from the given attribute.
- search() – This will search the entire subtree under the given directory entry for the given attributes. A LDAP search filter is supplied to determine which entries are included in the set of entries searched.

4.8 User Management

In order to use the system, users first need to register certain information with the MetaOS. Given a simple API to do this, ”wizards” can be written to guide the user through the registration step when the MetaOS is first customized.

4.8.1 User.class

The User class uses the methods of the DirLookup class to provide methods for easily changing a user's information in the LDAP directory. When instantiating a User object, the only argument is a globally unique mail address to use as the user ID. The User class then provides methods to query and modify the following attributes of the user's directory entry:

- user ID
- the user's full name
- user password
- description of the user, which can be used for arbitrary purposes
- user preferences

4.8.2 UserAdmin.class

The UserAdmin class uses the JNDI in order to create or remove user entries in the LDAP directory. The object classes AmbientComputingPerson and AmbientComputingDevice as defined in the schema above require specific attributes to be added to the entry when it is created. The UserAdmin class uses the LDAP configuration file to create a connection with the LDAP directory, then interacts with it using the following methods:

- listUsers() – list all users in the directory.
- createUser() – creates a user entry with the given user ID, password, full name, and last name.

- removeUser() – removes the given user entry from the directory.

In this way, the UserAdmin class can be used by administration tools separate from the MetaOS to facilitate integration with other systems.

4.9 Preference Management

4.9.1 User Preferences

The user preferences are stored in the AmbientComputingPrefs attribute of each user. This is a multi-valued attribute, as all preferences are stored here. They are sorted using the XResources style of preferences, where each property of an X object is specified as an increasingly specific property of a parent object or graph of objects. An example of this follows.

```
Chooser*geometry:               700x500+300+200
Chooser*allowShellResize:       false
Chooser*viewport.forceBars:     true
```

The preferences are stored in a similar fashion.

```
AmbientComputingPrefs: xmms.client.linux=/usr/bin/xmms
AmbientComputingPrefs: xmms.client.windows=
c:\Program Files\Winamp\Winamp.exe
AmbientComputingPrefs: xmms.volume=90%
AmbientComputingPrefs: xmms.playlist.rock=Tool.m3u
AmbientComputingPrefs: xmms.playlist.electronic=Orbital.m3u
```

User preferences are stored, queried and manipulated by the use of the following methods in the User class:

- getPrefs() – returns a list of preferences that match the given regular expression.

- setPref() – sets the value of the given preference to the new value.

- addPref() – adds the given preference to the list of this user's preferences.
- deletePref() – deletes the specified preference.
- deletePrefs() – deletes all preferences from this user's directory entry.

Menu properties are also stored in this fashion, allowing a menu to be customized to the user's tastes.

4.9.2 Device Preferences

Device preferences are stored in the same way as user preferences. Default device preferences can be inserted into the LDAP directory when a device is registered with the MetaOS system for the first time, in much the same way as initialization files in Windows. An example would be the controls for a refrigerator:

```
AmbientComputingPrefs: fridge.crisper.temp=5
AmbientComputingPrefs: fridge.freezer.temp=1
```

4.10 XML-RPC Interface

To facilitate development of devices to use with the MetaOS system, it became necessary to provide interfaces in other languages. Since the MetaOS messages are XML, other APIs can be written to generate and parse these messages to emulate the message calls in the Java source of the MetaOS.

An XML-RPC interface allows clients in other languages to issue XML messages to emulate devices. The Simple Object Access Protocol (SOAP) [2] specifies a standard for issuing remote calls with well-defined data types using the HTTP protocol. SOAP clients send requests to an HTTP server, which passes the request to a SOAP han-

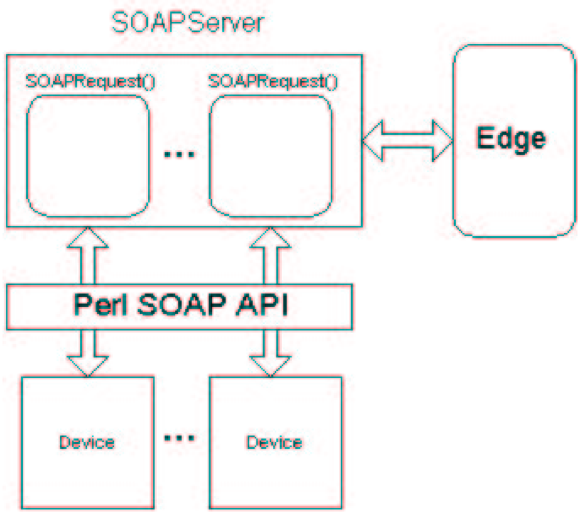

Figure 4.7: MetaOS SOAP Infrastructure

dler. The SOAP handler performs the specified operations, sending back a standard response. This follows the traditional request-response model of web servers.

The SOAPServer device pictured in Figure 4.7 implements the basics of a multi-threaded web server handling POST requests. The MetaOS SOAP server processes remote calls to one given method. The arguments sent by SOAP clients must be the body of a MetaOS message as defined earlier in this chapter. The SOAPServer device receives the request, reads the MetaOS message from the SOAP data, and adds the correct headers, specifying itself as the source address of the new MetaOS message. It then injects the message into the system and waits for the reply. When the reply is received, the body of the MetaOS message is placed into the SOAP reply headers and sent back to the SOAP client.

The SOAP::Lite toolkit [18] for Perl is a collection of Perl modules which provides a simple and lightweight interface to SOAP on both on the client and server side. Perl's penchant for quick scripting, as well as its popularity, were reasons for implementing the MetaOS SOAP interface using it. A Perl module was written using the SOAP::Lite toolkit to define an API similar to that of the Java API of the MetaOS system, which is listed below:

- registration messages, such as registerDevice(), registerEvent(), registerAction() and registerTodoItem()
- event messages - sendEvent()
- action messages - sendAction()
- informational messages, such as getInfo(), addInfo(), delInfo(), delAllInfo(), modInfo() and getList()
- miscellaneous messages to handle replies, such as parseACK(), parseLDAPresponse(), parseActionMsg() and parseListMessage()

Also, for clients using this interface to send actions, the getAction() subroutine is used to issue the *deviceID* messages as described in section 4.3.5.

Perl clients can then include the module and emulate a MetaOS device. This has allowed for rapid writing of devices as well as integration with CGI Perl scripts.

Chapter 5

Results

During development, a debugging class was written allowing debug messages to be written to a given file in the Unix syslog format. These messages are used to show the system functioning correctly in the examples below.

5.1 Hardware and Software Requirements

The x86-based PC has become the standard architecture of today, so it makes sense to develop the MetaOS with this architecture in mind. The implementation described in Chapter 4 runs on the following hardware and software settings. Both edges and hubs can be run on these settings.

- Pentium II 233 MHz and above
- 128 MB RAM and above
- Linux 2.4 series kernel or Windows 98/2000
- Sun Java 2 SDK, version 1.3.1

- Ethernet or 802.11b interface

Depending on the customization of the edge, the following items may also be used.

- IBM ViaVoice voice recognition software
- IBM ViaVoice Outloud speech software
- mp3 players such as mpg123 or Winamp
- Sound Blaster Live! sound card and speakers
- X10 modules

Embedded systems are another important architecture that the MetaOS should utilize, as their small hardware requirements and low cost allow them to be easily integrated with many consumer products.

Development in this area has used a RabbitCore RCM2100 Ethernet core module [22] from Rabbit Semiconductors. The RCM2100 RabbitCore contains a Rabbit microprocessor, integrated Ethernet port, and a royalty-free TCP/IP stack implementation. Development on the RabbitCore uses Dynamic C ®, which is included with developer boards. As of this writing, a MetaOS edge has been written, allowing for the creation of a device to blink LEDs on the development board. Other sensors and controls for environmental control are planned for the near future.

5.2 Message Transmission Architecture

The MetaOS system must use sockets to send messages using TCP/IP. The message transmission architecture has been throughly tested during development as it is a vital component of the MetaOS system. The debugging output from an edge joining the MetaOS domain was checked for correctness and is shown below.

The edge first instantiates a network interface thread which also creates the two helper threads.

```
7: 12:36:45 ==> Input and output streams created.
7: 12:36:45 ==> HEInterface:Reader.run() started
7: 12:36:45 ==> Writer::run() starting
7: 12:36:45 ==> Hub HEInterface started.
```

This initiates a socket connect to the hub's listen socket. When the hub accept()s this new socket, it instantiates its network interface thread.

```
7: 12:36:45 ==> Incoming socket connect from
                Socket[addr=localhost/127.0.0.1,
                port=1446,localport=6386]
7: 12:36:45 ==> Input and output streams created.
7: 12:36:46 ==> HEInterface:Reader.run() started
7: 12:36:46 ==> Writer::run() starting
```

After the edge side of the socket connection on the hub has been completed, the edge sends its routing information.

```
7: 12:36:46 ==> sending routing message:
0: 12:36:46 ==> <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message>
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1001785006028">
  <source_address>edge0</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.1850263360</message_ID>
  <identity>
    <user>HEInit</user>
    <group>nobody</group>
  </identity>
  <HEInit>
    <name>edge0</name>
  </HEInit>
</AmbComp:ambient_message>
```

The output below shows the correct behavior of the writer helper thread sending the routing message to the remote end of the socket stream.

```
7: 12:36:46 ==> HEInterface.send() starting
7: 12:36:46 ==> Writer::run() data removed from queue
7: 12:36:46 ==> Writer::run() data written to socket
7: 12:36:46 ==> HEInterface.send() finished
```

On the hub side, the routing information is read in. The hub creates the mapping between the socket the message was received on and the name sent in the message.

```
7: 12:36:46 ==> New route message read in for edge0
7: 12:36:47 ==> Message sent to edge0
```

Now, an entry in the hub's routing table exists for edge0, binding it to the address 127.0.0.1:1446. Then, an acknowledgement message is sent.

The routing information is sent and the edge then waits for confirmation from the hub.

```
7: 12:36:47 ==> HEInterface:Reader.run() data received from socket
7: 12:36:47 ==> Got Message from Hub:
0: 12:36:47 ==> <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message>
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1001785006642">
  <source_address>hub0</source_address>
  <destination_address>edge0</destination_address>
  <message_ID>edge0.1850263360</message_ID>
  <identity>
    <user>ack</user>
    <group>ack</group>
  </identity>
  <ack>
    <message_ID>edge0.1850263360</message_ID>
    <status>success</status>
    <data />
  </ack>
</AmbComp:ambient_message>
```

By using TCP/IP and sockets for connections between edges and hubs, we can use any data link layer protocol and transport medium that IP can use. During implementation, edges and hubs were connected using Ethernet, 802.11b wireless links, or both.

5.3 Messages

The XML messages in the MetaOS system are sent between hosts as text. Sending text messages allows other non-object based interfaces to connect to and interact with the MetaOS system easily, and XML messages provide necessary structure. The messages are then cast in JDOM ^{TM}[16] objects for manipulation in Java. Please see Appendices A and B for the message DTD and examples.

5.4 Device Interface

A common interface for device interaction with the MetaOS is required since there are so many different types of consumer devices. All devices communicate with the system using the methods defined in Section 4.5.1. The run() method of each device contains the device-specific logic.

5.5 Device Capabilities

Device capabilities must be learned dynamically as devices can enter and leave a MetaOS domain at any time. By using register messages, a device can dynamically enter the system and announce its capabilities to the hub.

5.5.1 Creation

Since the hub and edge processes define a Java main() method, they can be started like so:

```
java com.AmbientComputing.hub.Hub burns_hub_config.xml &
java com.AmbientComputing.edge.Edge burns_edge_config.xml &
```

Configuration file examples can be found in Appendix D.

The following lines indicate the hub process has created the event handler thread:

```
7: 14:19:55 ==> DeviceTree instantiated.
6: 14:19:55 ==> EventHandler thread started.
7: 14:19:55 ==> EventHandler: top of first if
```

While the hub is preparing for connections, the edge is also starting:

```
7: 14:20:06 ==> Writer::run() starting
7: 14:20:06 ==> HEInterface:Reader.run() started
7: 14:20:06 ==> Hub HEInterface started.
7: 14:20:06 ==> Top of Edge.run() while true loop
```

The edge has started its network interface thread and the two helper threads with it. When doing this, it has sent an *HEInit* message to identify this edge on the hub.

For the sake of brevity, the headers and footers of the messages in the following examples are omitted.

After the edge has completed its connection to the hub, the hub receives a *register_device* message from the new edge:

```
7: 14:20:06 ==> REGISTERING DEVICE...
0: 14:20:06 ==> <register>
    <identity>
      <user>lsanders</user>
      <group>adm</group>
    </identity>
    <device>
      <name>hub0.edge0</name>
      <ACL>110100100</ACL>
    </device>
  </register>
```

The edge then starts registering each device listed in its configuration file. For example, the SOAP interface is registered below:

```
7: 14:20:06 ==> REGISTERING DEVICE...
0: 14:20:06 ==> <register>
    <identity>
      <user>lsanders</user>
      <group>adm</group>
    </identity>
    <device>
      <name>hub0.edge0.SOAPServer</name>
      <ACL>110100100</ACL>
    </device>
  </register>
```

Devices, when instantiated, send messages to register events, actions, and todo list items in the device tree data structure, as seen below:

```
6: 14:20:07 ==> REGISTERING Action...
0: 14:20:07 ==>
  <source_address>hub0.edge0.Mp3Player0</source_address>
  <register>
    <identity>
      <user>larry</user>
      <group>adm</group>
    </identity>
    <action>
      <name>hub0.edge0.Mp3Player0.playURL</name>
      <ACL>101101101</ACL>
    </action>
  </register>

6: 14:20:07 ==> evtsDevice = hub0.edge0.Mp3Player0
6: 14:20:07 ==> actionName = hub0.edge0.Mp3Player0.playURL
7: 14:20:07 ==> Found Action Device Node: hub0.edge0.Mp3Player0
6: 14:20:07 ==> action added: hub0.edge0.Mp3Player0.playURL
```

The edge registers a X10 controller and an mp3 player in the same fashion. The debugging messages from this have been left out for the sake of brevity. When this is completed, the device tree data structure on the hub looks like the following, as show from the debug logs:

```
6: 14:20:07 ==> hub0
6: 14:20:07 ==> Id: root/adm
6: 14:20:07 ==> ACL: 110110100
6: 14:20:07 ==>   Events in this node:
6: 14:20:07 ==>   Actions in this node:
6: 14:20:07 ==> hub0.edge0
6: 14:20:07 ==> Id: lsanders/adm
6: 14:20:07 ==> ACL: 110100100
6: 14:20:07 ==>   Events in this node:
6: 14:20:07 ==>   Actions in this node:
6: 14:20:07 ==> hub0.edge0.Mp3Player0
6: 14:20:07 ==> Id: lsanders/adm
6: 14:20:07 ==> ACL: 110100100
6: 14:20:07 ==>   Events in this node:
```

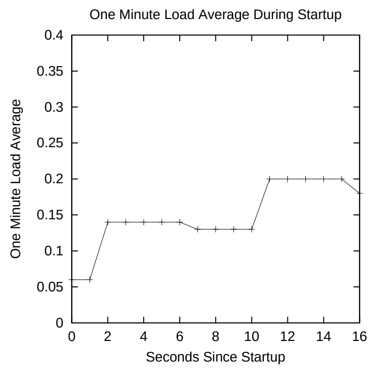

Figure 5.1: System Load During Startup

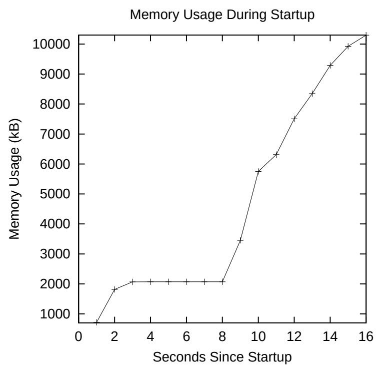

Figure 5.2: Memory Usage During Startup

```
6: 14:20:07 ==>   Actions in this node:
0: 14:20:07 ==>     action: hub0.edge0.Mp3Player0.playURL
                    (other actions removed for brevity)
6: 14:20:07 ==> hub0.edge0.SOAPServer
6: 14:20:07 ==> Id: lsanders/adm
6: 14:20:07 ==> ACL: 110100100
6: 14:20:07 ==>   Events in this node:
6: 14:20:07 ==>   Actions in this node:
6: 14:20:07 ==> hub0.edge0.X10
6: 14:20:07 ==> Id: lsanders/adm
6: 14:20:07 ==> ACL: 110100100
6: 14:20:07 ==>   Events in this node:
6: 14:20:07 ==>   Actions in this node:
0: 14:20:07 ==>     action: hub0.edge0.X10.coldroomLamp
0: 14:20:07 ==>     action: hub0.edge0.X10.light0
```

The two graphs in Figures 5.1 and 5.2 show the system load (number of waiting jobs) and memory usage of a hub and edge being executed on the same host in the manner just illustrated. The hub is started first, and the load and memory usage jumps as expected. When the edge is started ten seconds after the hub, at the eleven-second mark, the load and memory usage jumps again. However, the edge increases the memory usage by only four or five megabytes, which bodes well for scalability.

5.5.2 Discovery

Currently, devices can discover the capabilities of another device by sending a *list* message to the hub in order to determine what devices and actions have been registered with the hub. This message can then be parsed by the device.

A Perl shell was written as a development tool to allow developers to send actions to devices easily for device testing. Since it uses list messages to interact with the MetaOS, it can be used here to demonstrate device and action discovery.

The device subtype of the *list* message is sent to the hub first to discover what edges are connected:

```
[lsanders@ambient hub0]$ ls
essence_edge
gibson_edge
```

The cd command sets the requested root of the *list* message to the specified device, e.g. essence_edge. Another *list* device query is sent to determine the devices controlled by this edge.

```
[lsanders@ambient hub0]$ cd essence_edge
[lsanders@ambient hub0.essence_edge]$ ls
FrameGrabber0
Mp3Player0
Quotes
X10
```

A *list* action query is then sent to the hub to determine which actions the current device can execute.

```
[lsanders@ambient hub0.essence_edge]$ actions X10
hub0.essence_edge.X10.coldroomLamp
hub0.essence_edge.X10.light0
hub0.essence_edge.X10.coldroomLamp
hub0.essence_edge.X10.light0
```

Finally, an *action* message is sent to execute the specified action.

```
[lsanders@ambient hub0.essence_edge]$ sendAction X10 light0 on
Action hub0.essence_edge.X10.light0 on sent
```

5.6 Events

The triggering and handling of events is at the core of the MetaOS system. To verify that *event* and *action* messages work, a device can be registered quickly using the SOAP interface. For this example, a client connected to the SOAP interface will trigger an event, which will send actions to two different devices.

First (after the device is registered), a new event is registered.

```
  <register>
    <identity />
    <event>
      <name>hub0.edge0.SOAPServer.client0.event0</name>
      <ACL>111111111</ACL>
    </event>
  </register>
```

This event will trigger the playing of a mp3 from the network, and will turn on a light. These are todo list items of the event just registered.

```
  <todolistitem>
    <identity>
      <user>jdavis</user>
      <group>users</group>
    </identity>
    <parent_event>
  hub0.edge0.SOAPServer.client0.event0
</parent_event>
    <action>
      <name>hub0.edge0.Mp3Player0.playURL</name>
      <data>
        <arg>http://www.hithere.com/songs.m3u</arg>
      </data>
    </action>
  </todolistitem>
```

The second action is omitted here for sake of brevity. After these operations are completed, the device tree data structure contains entries for all three operations.

```
6: 18:42:17 ==> hub0.edge0.SOAPServer.client0
6: 18:42:17 ==> Id: null/null
6: 18:42:17 ==> ACL: 111111111
```

```
6: 18:42:17 ==>   Events in this node:
0: 18:42:17 ==>   event: hub0.edge0.SOAPServer.client0.event0
0: 18:42:17 ==>     actionName: hub.edge0.Mp3Player0.playURL
0: 18:42:17 ==>     identity: lsanders/adm
0: 18:42:17 ==>     actionName: hub.edge0.X10.light0
0: 18:42:17 ==>     identity: lsanders/adm
6: 18:42:17 ==>   Actions in this node:
```

Now, the event can be triggered by sending an *event* message to the hub.

```
<event>
  <name>hub0.edge0.SOAPServer.client0.event0</name>
  <data />
</event>
```

This sends *action* messages to the two devices in event0's todo list. This also shows that macros are implemented correctly.

```
<destination_address>
  hub0.edge0.Mp3Player0
</destination_address>
<identity>
  <user>lsanders</user>
<group>adm</group>
</identity>
<action>
  <name>hub0.edge0.Mp3Player0.playURL</name>
<data>
  <arg>http://www.hithere.com/songs.m3u</arg>
  </data>
</action>

<destination_address>hub0.edge0.X10</destination_address>
<identity>
  <user>lsanders</user>
<group>adm</group>
</identity>
<action>
  <name>hub0.edge0.X10.light0</name>
<data>
  <arg>on</arg>
  </data>
</action>
```

5.6.1 Preferences

The MetaOS LDAP interface controls the user's personal settings of the MetaOS, so failures in it will be quite noticable to the user. An LDAP server was filled with several user entries to provide data to modify; device entries are similar, so test modifications on them are not shown.

The following example shows a value being added to one of the user's attributes.

```
<info>
  <add>
    <name type="user">jdavis@ambientcomputing.com</name>
    <data>
      <param name="AmbientComputingprefs"
         value="mp3.player=/usr/bin/xmms" />
    </data>
  </add>
</info>
```

To show the state of the directory entry, the OpenLDAP suite of tools includes ldapsearch, a command line tool which outputs the directory entries matching its command line arguments in the LDIF format. Before the following message was sent to the hub, the user's entry looked like this:

```
dn: uid=jdavis@ambientcomputing.com,
    dc=users, dc=AmbientComputing, dc=com
objectClass: AmbientComputingPerson
sn: Davis
cn: Jesse "Iceweasel" Davis
uid: jdavis@ambientcomputing.com
userPassword:: dGhpc2lzbXlwYXNzd29yZA==
AmbientComputingPrefs: mp3.volume=80%
AmbientComputingPrefs: TV.channel0=ESPN
```

After the add operation completes, the value is in the LDAP directory.

```
uid: jdavis@ambientcomputing.com
userPassword:: dGhpc2lzbXlwYXNzd29yZA==
AmbientComputingPrefs: mp3.volume=80%
AmbientComputingPrefs: TV.channel0=ESPN
AmbientComputingPrefs: mp3.player=/usr/bin/xmms
```

We can delete one value from any attribute,

```
<info>
  <delete type="value">
    <name type="user">jdavis@ambientcomputing.com</name>
    <data>
      <param name="AmbientComputingprefs"
         value="mp3.player=/usr/bin/xmms" />
    </data>
  </delete>
</info>
```

which results in the following LDAP entry.

```
uid: jdavis@ambientcomputing.com
userPassword:: dGhpc2lzbXlwYXNzd29yZA==
AmbientComputingPrefs: mp3.volume=80%
AmbientComputingPrefs: TV.channel0=ESPN
```

All attributes can also be deleted,

```
<info>
  <delete type="attribute">
    <name type="user">jdavis@ambientcomputing.com</name>
    <data>
      <param name="AmbientComputingprefs" value="" />
    </data>
  </delete>
</info>
changing the LDAP entry as shown below.
uid: jdavis@ambientcomputing.com
userPassword:: dGhpc2lzbXlwYXNzd29yZA==
```

Single values can be modified as well with the *modify* message type.

```
<info>
  <modify>
    <name type="user">jdavis@ambientcomputing.com</name>
    <data>
      <param name="AmbientComputingprefs"
             value="mp3.player=/usr/bin/xmms" />
      <param name="AmbientComputingprefs"
             value="mp3.player=c\:Program Files\Winamp\Winamp.exe" />
    </data>
  </modify>
</info>
```

The LDAP directory entry then changes as shown.

```
AmbientComputingPrefs: mp3.volume=80%
AmbientComputingPrefs: mp3.player=/usr/bin/xmms

AmbientComputingPrefs: mp3.volume=80%
AmbientComputingPrefs: mp3.player=
c\:Program Files\Winamp\Winamp.exe
```

The *query* informational message can be used to find the values of the specified LDAP entry. The type attribute specifies what type of object the entry is. If a wildcard is specified, as depicted, we can return back all entries that have values for the specified attributes. For example, sending the *query* message below:

```
  <info>
    <query>
      <name type="user">*</name>
      <data>
        <param name="sn" value="" />
        <param name="cn" value="" />
      </data>
    </query>
  </info>
```

sends back the following *result* message.

```
  <info>
    <response>
      <message_ID type="">edge0.267726262</message_ID>
      <data>
        <result name="jdavis@ambientcomputing.com">
          <param name="sn" value="Davis" />
          <param name="cn" value="Jesse Davis" />
        </result>
        <result name="uid=lsanders@ambientcomputing.com">
          <param name="sn" value="Sanders" />
          <param name="cn" value="Larry Sanders" />
        </result>
      </data>
    </response>
  </info>
```

5.7 User Identification

User differentiation is accomplished by changing the <identity> element of most messages to reflect the user and group sending those messages. Users can currently specify themselves by logging into the web interface. Support for other authentication schemes is straightforward.

5.8 Security

User information must be kept secure so that user and system preferences are not tampered with. The <identity> element specifies the user and group identities in a message. The <acl> element specifies the permissions associated with the message. These are checked against the user, group and permissions of the device tree entry that the operation is working on.

An example of this occurs during device registration:

```
7: 14:20:06 ==> REGISTERING DEVICE...
7: 14:20:06 ==> deviceName = hub0.edge0.SOAPServer*
7: 14:20:06 ==> Identity = lsanders/adm
7: 14:20:06 ==> DeviceTree insert: Checking permissions
7: 14:20:06 ==> DeviceTree insert: inserting ID=lsanders/adm
7: 14:20:06 ==> DeviceTree insert: root ID =lsanders/adm
7: 14:20:06 ==> Woohoo! User can insert!
6: 14:20:06 ==> device added: hub0.edge0.SOAPServer
7: 14:20:06 ==> deviceTree is now:
                (removed for brevity)
```

The user and group name match, so the addition of the device with the specified owner and group is allowed. If these fail, the device addition is denied, as in the following example.

```
7: 14:20:10 ==> REGISTERING DEVICE...
7: 14:20:10 ==> deviceName = hub0.edge0.SOAPServer*
7: 14:20:10 ==> Identity = jdavis/users
7: 14:20:10 ==> DeviceTree insert: Checking permissions
```

```
7: 14:20:10 ==> DeviceTree insert: inserting ID=jdavis/users
7: 14:20:10 ==> DeviceTree insert: root ID =lsanders/adm
7: 14:20:10 ==> Permission denied.
```

5.9 SOAP Interface

The SOAP interface to the MetaOS is modeled after a multi-threaded HTTP server that handles POST requests. First the SOAPServer device is started.

```
6: 13:47:43 ==> SOAPServer device started.
7: 13:47:43 ==> SOAPServer accepting connections on port 8083
```

The web server waits for connections. Once a connection is created, the HTTP header must be read in.

```
7: 14:00:04 ==> line = POST / HTTP/1.0
7: 14:00:05 ==> line = Accept: text/xml
7: 14:00:05 ==> line = Accept: multipart/*
(extra lines in header removed for space)
```

The body of the HTTP request is then read in.

```
7: 14:00:05 ==> SOAP message from client:
7: 14:00:05 ==> <?xml version="1.0" encoding="UTF-8"?>
  <SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
      <AmbientMessage>
        <c-gensym5
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
            xsi:type="SOAP-ENC:base64">
          PD48aWRlbnRpdHk+PHVzZXI+YmV3eTwvdXNlcj48Z3JvdXA+YWRtPC
          9ncm91cD48L2lkZW50aXR5PjxpbmZvPjxxdWVyeT48bmFtZSB0eXBl
          PSJ1c2VyIj4qPC9uYW1lPjxkYXRhPjxwYXJhbSBuYW1lPSIqIiB2YW
          x1ZT0iIj48L3BhcmFtPjwvZGF0YT48L3F1ZXJ5PjwvaW5mbz48Lz4=
        </c-gensym5>
      </AmbientMessage>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

The body of the SOAP message is base-64 encoded to make the parsing more efficient with the Perl SOAP::Lite module. The body is then decoded to the following MetaOS message body:

```
7: Sat Jun 30 14:00:05 CDT 2001 ==> new msg body =
  <identity>
    <user>jdavis</user>
<group>adm</group>
  </identity>
  <info>
    <query>
      <name type="user">*</name>
  <data>
    <param name="*" value=""></param>
  </data>
    </query>
  </info>
```

This body is given the correct headers so that the response will return back to the SOAPServer device. This is then handed to the correct thread in the SOAPServer device, which sends the response back to the client:

```
7: 14:00:05 ==> outgoing message follows:
7: 14:00:05 ==> HTTP/1.1 200 OK
Date: Sep Sep 29 14:00:05 CDT 2001
Server: AmbCompSOAPServer/0.0.1
Content-Length: 1901
Content-Type: text/xml
SOAPServer: AmbCompSOAPServer/Java/0.0.1

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"
  xsi="http://www.w3.org/1999/XMLSchema-instance"
  xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <namesp1:AmbientMessageResponse xmlns:namesp1="World">
      <s-gensym5 type="xsd:string">
      (body removed for space)
      </s-gensym5>
    </namesp1:AmbientMessageResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The Perl API then parses the response and places the necessary data into strings or hashes as defined by the API.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The project designed, constructed, and verified the execution of an ambient computing system. All goals of the basic architecture were realized successfully. The following was determined:

- The application of the meta-operating system idea to an ambient computing system seemed to be a natural fit. Events and actions were modeled in a similar way to the traditional operating system, which helped conceptually during development. Devices were easier to write once the device driver model was used.
- The message architecture system caused no blocking and delivered messages without any errors. The load on the system imposed by it was also found to be minimal.
- The preferences architecture is adequate to personalize all objects in the MetaOS system.
- The SOAP interface and corresponding Perl API have proven to be of great use in quickly creating devices with Perl, allowing developers to use Perl's large library of software with the MetaOS system.

- Basic access control lists based on the Unix permissions system are adequate for controlling access to MetaOS objects.

6.2 Future Work

The project was successfully implemented, but improvements can always be made. The following is a list of tasks that could improve the MetaOS system and any future versions of it.

- An SLP interface with the MetaOS system could help integrate service discovery within the MetaOS with enterprise-level system discovery. A preliminary version of this interface has been written.
- A secure protocol and mechanism for hub-to-hub communication is necessary to expand the meta-operating system idea to multiple domains, as well as to provide replication abilities.
- SSL connections should be implemented on the network interface threads, as well as on the connection to the LDAP directory.
- Extended access control lists which could filter on the value of any attribute in the database or device tree data structure would provide finer-grain control than the access control lists in place now.
- A UPnP interface to the MetaOS will be very beneficial, as communicating with Microsoft's .NET system will no doubt become important in the near future.
- A JINI ^{TM}[27] interface for sending messages could be helpful in interfacing the MetaOS with software written with Java's enterprise-level APIs. JINI provides interfaces for sending Java objects through different protocols on a network.

Appendix A

Message Document Type Definitions

```
<!ELEMENT source_address (#PCDATA)>
<!ELEMENT destination_address (#PCDATA)>
<!ELEMENT message_ID (#PCDATA)>
<!ELEMENT user (#PCDATA)>
<!ELEMENT group (#PCDATA)>
<!ELEMENT identity (user, group)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT acl (#PCDATA)>
<!ELEMENT status (#PCDATA)>

<!ELEMENT param EMPTY>
<!ATTLIST param
    name CDATA #REQUIRED
    value CDATA #REQUIRED
>
<!ELEMENT ldapdata (param*)>
<!ELEMENT ldapdata1 (param)>
<!ELEMENT ldapdata2 (param, param)>
<!ELEMENT responsedata (result*)>
<!ELEMENT result (param*)>
<!ATTLIST result
    name CDATA #REQUIRED
>

<!ELEMENT arg (#PCDATA)>
a
<!ELEMENT data (arg*)>
<!ELEMENT data1 (arg)>
<!ELEMENT data2 (arg, arg)>
<!ELEMENT data3 (arg, arg, arg)>
```

```
<!ELEMENT getDeviceID (identity, ldapdata1)>

<!ELEMENT AmbComp:ambient_message
          ((source_address, destination_address, message_ID),
           (identity, ack),
           (identity, nak),
           (identity, HEInit),
           (register),
           (identity, todolistitem),
           (identity, event),
           (identity, action, calling_event),
           (getDevideID),
           (identity, info),
           (list),
           (deviceEntry),
           (actions))
>
<!ATTLIST AmbComp:ambient_message
    xmlns:AmbComp CDATA #REQUIRED
    version CDATA #REQUIRED
    message_time CDATA #REQUIRED
>

<!-- status here is "success" -->
<!ELEMENT ack (message_ID, status, data)>

<!-- status here is "failure" -->
<!ELEMENT nak (message_ID, status, data)>

<!ELEMENT HEInit (name)>

<!ELEMENT register
          (identity, (device|register_event|register_action))>
<!ELEMENT device (name, acl)>
<!ELEMENT register_event (name, acl)>
<!ELEMENT register_action (name, acl)>

<!ELEMENT parent_event (#PCDATA)>
<!ELEMENT todolistitem (identity, parent_event, action)>

<!ELEMENT event (name, data)>

<!ELEMENT action (name, data)>
<!ELEMENT calling_event (name, data)>

<!ELEMENT info (query|response|add|modify|delete)>
<!ELEMENT info-name (#PCDATA)>
<!ATTLIST info-name
```

```
    type CDATA #REQUIRED
>
<!ELEMENT query (info-name, ldapdata)>
<!ELEMENT response (message_ID, responsedata)>
<!ELEMENT add (info-name, ldapdata1)>
<!ELEMENT modify (info-name, ldapdata2)>
<!ELEMENT delete (info-name, ldapdata1)>
<!ATTLIST delete
    type (value|attribute) #REQUIRED
>

<!ELEMENT listType (#PCDATA)>
<!ELEMENT root (#PCDATA)>
<!ELEMENT device (#PCDATA)>
<!ELEMENT action (#PCDATA)>

<!-- listType should be "device" or "actions" -->
<!ELEMENT list (listType, root, identity)>

<!ELEMENT deviceEntry (device*)>

<!ELEMENT actions (action*)>
```

Appendix B

XML Message Examples

B.1 Registration Messages

B.1.1 register - device

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322259806">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.316394043</message_ID>
  <register>
   <identity>
      <user>lsanders</user>
      <group>adm</group>
</identity>
<device>
  <name>hub0.edge0.SOAPServer.client1</name>
  <ACL>111111111</ACL>
</device>
   </register>
</AmbComp:ambient_message>
```

B.1.2 register - event

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322259806">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.316394043</message_ID>
  <register>
   <identity>
      <user>lsanders</user>
      <group>adm</group>
</identity>
<event>
  <name>hub0.edge0.SOAPServer.client1.event0</name>
  <ACL>111111111</ACL>
</event>
   </register>
</AmbComp:ambient_message>
```

B.1.3 register - action

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322260330">
  <source_address>
    hub0.edge0.SOAPServer.client1
  </source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.116832368</message_ID>
  <register>
   <identity>
      <user>lsanders</user>
      <group>adm</group>
</identity>
<action>
   <name>hub0.edge0.SOAPServer.client1.sayHi</name>
   <ACL>111111111</ACL>
</action>
  </register>
</AmbComp:ambient_message>
```

B.1.4 register - todo list item

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322259806">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.316394043</message_ID>
  <identity>
    <user>lsanders</user>
<group>adm</group>
  </identity>
  <todolistitem>
    <identity>
  <user>jdavis</user>
  <group>users</group>
</identity>
<parent_event>event0</parent_event>
<action>
  <name>action0</name>
  <data>
    <arg>userPassword</arg>
    <arg>howdy</arg>
  </data>
</action>
  </todolistitem>
</AmbComp:ambient_message>
```

B.1.5 HEInit

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322259806">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.316394043</message_ID>
  <identity>
    <user>lsanders</user>
<group>adm</group>
  </identity>
  <HEInit>
    <name>edge0</name>
  </HEInit>
</AmbComp:ambient_message>
```

B.2 Event Messages

B.2.1 event

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322259806">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.316394043</message_ID>
  <identity>
    <user>lsanders</user>
<group>adm</group>
  </identity>
  <event>
    <name>event0</name>
<data>
  <arg>userPassword</arg>
  <arg>howdy</arg>
</data>
  </event>
</AmbComp:ambient_message>
```

B.3 Action Messages

B.3.1 action

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="995322271580">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.edge0.SOAPServer.client0
  </destination_address>
  <message_ID>edge0.1173339101</message_ID>
  <identity>
    <user>lsanders</user>
    <group>adm</group>
  </identity>
  <action>
    <name>hub0.edge0.SOAPServer.client0.sayHi</name>
    <data>
      <arg>there</arg>
      <arg>pardner</arg>
    </data>
  </action>
  <calling_event>
    <name>null:hub0.edge0.SOAPServer</name>
    <data />
  </calling_event>
</AmbComp:ambient_message>
```

B.4 Informational Messages

B.4.1 query

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1000497546057">
  <source_address>hub0.edge0.SOAPServer</source_address>
  <destination_address>hub0</destination_address>
  <message_ID>edge0.267726262</message_ID>
  <identity>
    <user>lsanders</user>
    <group>adm</group>
  </identity>
  <info>
    <query>
      <name type="user">bewy@ambientcomputing.com</name>
      <data>
        <param name="cn" value="" />
        <param name="sn" value="" />
      </data>
    </query>
  </info>
</AmbComp:ambient_message>
```

B.4.2 response

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1000497546255">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.edge0.SOAPServer
  </destination_address>
  <message_ID>edge0.267726262</message_ID>
  <identity>
    <user>ldap</user>
    <group>ldap</group>
  </identity>
  <info>
    <response>
      <message_ID type="">edge0.267726262</message_ID>
      <data>
        <result name="uid=bewy@ambientcomputing.com">
          <param name="sn" value="Ewy" />
          <param name="cn" value="Ben Ewy" />
        </result>
      </data>
    </response>
  </info>
</AmbComp:ambient_message>
```

B.4.3 add

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1000497546255">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.edge0.SOAPServer
  </destination_address>
  <message_ID>edge0.267726262</message_ID>
  <identity>
    <user>ldap</user>
    <group>ldap</group>
  </identity>
  <info>
    <add>
      <name type="user">buchanan@ambientcomputing.com</name>
      <data>
        <param name="AmbientComputingprefs"
         value="mp3.player=/usr/bin/xmms" />
      </data>
    </add>
  </info>
```

```
</AmbComp:ambient_message>
```

B.4.4 delete - value

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1000497546255">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.edge0.SOAPServer
  </destination_address>
  <message_ID>edge0.267726262</message_ID>
  <identity>
    <user>ldap</user>
    <group>ldap</group>
  </identity>
  <info>
    <delete type="value">
      <name type="user">buchanan@ambientcomputing.com</name>
      <data>
        <param name="AmbientComputingprefs"
       value="mp3.player=/usr/bin/xmms" />
      </data>
    </delete>
  </info>
</AmbComp:ambient_message>
```

B.4.5 delete - attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1000497546255">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.edge0.SOAPServer
  </destination_address>
  <message_ID>edge0.267726262</message_ID>
  <identity>
    <user>ldap</user>
    <group>ldap</group>
  </identity>
  <info>
```

```
    <delete type="attribute">
      <name type="user">buchanan@ambientcomputing.com</name>
      <data>
        <param name="AmbientComputingprefs" value="" />
      </data>
    </delete>
  </info>
</AmbComp:ambient_message>
```

B.4.6 modify

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
    version="1.0" message_time="1000497546255">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.edge0.SOAPServer
  </destination_address>
  <message_ID>edge0.267726262</message_ID>
  <identity>
    <user>ldap</user>
    <group>ldap</group>
  </identity>
  <info>
    <modify>
      <name type="user">buchanan@ambientcomputing.com</name>
      <data>
        <param name="AmbientComputingprefs"
       value="/usr/bin/xmms" />
        <param name="AmbientComputingprefs"
       value="/usr/bin/mpg123" />
      </data>
    </modify>
  </info>
</AmbComp:ambient_message>
```

B.4.7 list - device query

```
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="1000330185156">
  <source_address>
    hub0.essence_edge.SOAPServer
  </source_address>
  <destination_address>hub0</destination_address>
  <message_ID>essence_edge.900376418</message_ID>
  <list>
    <listType>device</listType>
    <root>hub0.essence_edge</root>
    <identity />
  </list>
</AmbComp:ambient_message>
```

B.4.8 list - device response

```
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="1000330185165">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.essence_edge.SOAPServer
  </destination_address>
  <message_ID>essence_edge.900376418</message_ID>
  <deviceEntry>
    <device>hub0.essence_edge.FrameGrabber0</device>
    <device>hub0.essence_edge.Mp3Player0</device>
    <device>hub0.essence_edge.Quotes</device>
    <device>hub0.essence_edge.SOAPServer</device>
    <device>hub0.essence_edge.TextToSpeech</device>
    <device>hub0.essence_edge.VoicePreferences</device>
    <device>hub0.essence_edge.VoiceRec</device>
    <device>hub0.essence_edge.X10</device>
    <device>hub0.essence_edge.X10Wireless</device>
    <device>hub0.essence_edge.test</device>
  </deviceEntry>
</AmbComp:ambient_message>
```

B.4.9 list - action query

```
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="1000330615470">
  <source_address>
    hub0.essence_edge.SOAPServer
  </source_address>
  <destination_address>hub0</destination_address>
  <message_ID>essence_edge.2081776766</message_ID>
  <list>
    <listType>actions</listType>
    <root>hub0.essence_edge.FrameGrabber0</root>
    <identity />
  </list>
</AmbComp:ambient_message>
```

B.4.10 list - action response

```
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="1000330615482">
  <source_address>hub0</source_address>
  <destination_address>
    hub0.essence_edge.SOAPServer
  </destination_address>
  <message_ID>essence_edge.2081776766</message_ID>
  <actions>
    <action>
  hub0.essence_edge.FrameGrabber0.grabFrames
</action>
  </actions>
</AmbComp:ambient_message>
```

B.5 Miscellaneous Messages

B.5.1 ack

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="1000496786930">
  <source_address>hub0</source_address>
  <destination_address>hub0.edge0</destination_address>
  <message_ID>edge0.710364944</message_ID>
  <identity>
    <user>root</user>
    <group>adm</group>
  </identity>
  <ack>
    <message_ID>edge0.710364944</message_ID>
    <status>success</status>
    <data />
  </ack>
</AmbComp:ambient_message>
```

B.5.2 nak

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="1000496786930">
  <source_address>hub0</source_address>
  <destination_address>hub0.edge0</destination_address>
  <message_ID>edge0.710364944</message_ID>
  <identity>
    <user>root</user>
    <group>adm</group>
  </identity>
  <ack>
    <message_ID>edge0.710364944</message_ID>
    <status>failure</status>
    <data param="error_string"
      value="This is an error string."/>
    <data param="error_code" value="42"/>
  </ack>
</AmbComp:ambient_message>
```

B.5.3 getDeviceID - query

```
<?xml version="1.0" encoding= "UTF-8"?>
<getDeviceID>
  <identity>
    <user>lsanders</user>
    <group>adm</group>
  </identity>
  <data>
    <param name="port" value="9999" />
  </data>
</getDeviceID>
```

B.5.4 getDeviceID - response

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AmbComp:ambient_message
  SYSTEM "DTDs/ambient_message.dtd">
<AmbComp:ambient_message
    xmlns:AmbComp="http://www.ambientcomputing.com"
version="1.0" message_time="995322259615">
```

```
  <source_address>edge0.SOAPServer</source_address>
  <destination_address>
    hub0.edge0.SOAPServer.client1
  </destination_address>
  <message_ID>edge0.1962494861</message_ID>
  <getDeviceID>
    <data>
      <param name="deviceID"
         value="hub0.edge0.SOAPServer.client1" />
    </data>
  </getDeviceID>
</AmbComp:ambient_message>
```

Appendix C

MetaOS LDAP Schema

```
#
# Ambient Computing’s directory schema items
#
# depends upon:
#     core.schema
#     inetorgperson.schema
#
# These are provided for informational purposes only.
# Jesse Davis (jdavis@ambientcomputing.com)  April 11, 2001

# Case is significant.
attributetype ( 1.3.6.1.4.1.8733.1.1.1
    NAME ’AmbientComputingPrefs’
    DESC ’Ambient Computing preferences object’
    EQUALITY caseExactIA5Match
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

attributetype ( 1.3.6.1.4.1.8733.1.2.2 NAME ’location’
    DESC ’Ambient Computing Device location’
    EQUALITY caseExactIA5Match
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

objectClass ( 1.3.6.1.4.1.8733.1.1
    NAME ’AmbientComputingPerson’
    DESC ’Ambient Computing User Object’
    SUP ( person $ organizationalPerson $ inetOrgPerson )
    MUST ( cn $ sn $ uid $ userPassword )
    MAY ( AmbientComputingPrefs $ location) )

attributetype ( 1.3.6.1.4.1.8733.1.2.3 NAME ’state’
    DESC ’Ambient Computing Device state’
    EQUALITY caseExactIA5Match
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

attributetype ( 1.3.6.1.4.1.8733.1.2.1 NAME ’deviceType’
    DESC ’Ambient Computing Device type description’
    EQUALITY caseExactIA5Match
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

objectClass ( 1.3.6.1.4.1.8733.1.2
    NAME ’AmbientComputingDevice’
    DESC ’Ambient Computing Device Object’
    SUP ( top )
    MUST ( cn $ deviceType )
    MAY ( state $ location $ description $ displayName $
          AmbientComputingPrefs ) )
```

Appendix D

Sample Configuration Files

D.1 Hub

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DOCTYPE AmbComp:ambient_message
     SYSTEM "DTDs/hub_config.dtd" -->
<AmbComp:hub_config
    xmlns:AmbComp="http://www.ambientcomputing.com">

    <!-- need to create DTD for this -->
    <!-- add DOCTYPE back in when we do -->
    <!-- do we want version or modification-time -->
    <!--   attributes in level element -->

    <AmbComp:description>
        This is the configuration file for the hub process.
    </AmbComp:description>

    <!-- Connection specific info -->
        <!-- port to create listen socket on -->
    <AmbComp:port>6386</AmbComp:port>
        <!-- encryption type for socket -->
    <AmbComp:encryption>none</AmbComp:encryption>

    <!-- Device specific info -->
    <AmbComp:device-prefix>hub0</AmbComp:device-prefix>

    <!-- logfile  this is fully qualified -->
    <AmbComp:logfile>
    /usr/local/ambient/logs/hub.log
</AmbComp:logfile>

</AmbComp:hub_config>
```

D.2 Edge

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DOCTYPE AmbComp:ambient_message
     SYSTEM "DTDs/edge_config.dtd" -->
<AmbComp:edge_config
    xmlns:AmbComp="http://www.ambientcomputing.com">

    <!-- need to create DTD for this -->
    <!-- add DOCTYPE back in above when we do -->
    <!-- do we want version or modification-time -->
    <!--   attributes in level element -->

    <AmbComp:description>
        This is the configuration file for the edge process.
    </AmbComp:description>

    <!-- Connection specific info -->
        <!-- address of hub -->
    <AmbComp:address>127.0.0.1</AmbComp:address>
        <!-- port to connect to -->
    <AmbComp:port>6386</AmbComp:port>
        <!-- encryption type for socket -->
    <AmbComp:encryption>none</AmbComp:encryption>
        <!-- The Name of the hub we belong to -->
    <AmbComp:hub-name>hub0</AmbComp:hub-name>
        <!-- full path to logfile -->
    <AmbComp:logfile>
    /usr/local/ambient/logs/edge.log
</AmbComp:logfile>

    <!-- Device specific info -->
    <AmbComp:device-prefix>edge0</AmbComp:device-prefix>

    <!-- Devices below this -->
    <devices>
<device>
            <name>Mp3Player0</name>
            <handler>Mp3Player</handler>
        </device>
        <device>
            <name>X10</name>
            <handler>X10Device</handler>
        </device>
        <device>
            <name>X10</name>
            <handler>X10Device</handler>
        </device>
```

```
<device>
            <name>SOAPServer</name>
            <handler>SOAPServer</handler>
        </device>
    </devices>

    <!-- may want to have xml doc for each device -->
    <!--   possibly have each device have own parser -->
</AmbComp:edge_config>
```

D.3 LDAP

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DOCTYPE AmbComp:ambient_message
     SYSTEM "DTDs/ldap_config.dtd" -->
<AmbComp:ldap_config
    xmlns:AmbComp="http://www.ambientcomputing.com">

    <!-- need to create DTD for this -->
    <!-- add DOCTYPE back in when we do -->
    <!-- do we want version or modification-time -->
    <!--   attributes in level element -->

    <AmbComp:description>
        This is the configuration file for the classes
managing LDAP database connections in the Ambient system.
    </AmbComp:description>

    <!-- Connection specific info -->
        <!-- initial context factory -->
    <AmbComp:initial_context_factory>
        com.sun.jndi.ldap.LdapCtxFactory
    </AmbComp:initial_context_factory>
        <!-- URL of LDAP server root -->
    <AmbComp:root_url>
      ldap://frodo.ittc.ku.edu:389/dc=ambientcomputing,dc=com
    </AmbComp:root_url>
        <!-- security authentication type -->
        <!-- may want to subtag for different type -->
    <AmbComp:security_auth>simple</AmbComp:security_auth>
        <!-- security principal -->
    <AmbComp:security_principal>
        cn=root,dc=AmbientComputing,dc=com
    </AmbComp:security_principal>
        <!-- password for server -->
    <AmbComp:password>i’m not telling</AmbComp:password>
</AmbComp:ldap_config>
```

Bibliography

[1] Bill Dirks. Video For Linux Two. http://www.thedirks.org/v4l2/, 2001.

[2] Don Box et al. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/, May 2000.

[3] Brett Becker and others. Implementation of Ambient Computational Environments Concepts in Microsoft Windows. Technical Report ITTC-FY2002-22731-01, University of Kansas Information and Telecommunication Technology Center, August 2001.

[4] Hewlett-Packard Company. Chai Appliance Platform. http://www.hp.com/products1/embedded/whatischai.html, 1994-2001.

[5] Microsoft Corporation. Universal Plug and Play Device Architecture. http://www.upnp.org/download/UPnPDA10_20000613.htm, 1999-2000.

[6] Microsoft Corporation. Microsoft .NET. http://www.microsoft.com/net/, 2001.

[7] Dallas Semiconductor Corporation. *Book of iButton Standards*, 2001.

[8] OpenLDAP Foundation. Introduction to OpenLDAP Directory Services. http://www.openldap.org/devel/admin/intro.html, 2001.

[9] OpenLDAP Foundation. OpenLDAP: Community Developed LDAP Software. http://www.openldap.org/, 2001.

[10] Steven D. Gribble et al. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks*, 35(4):473–497, 2001.

[11] Al Grimstad et al. Naming Plan for Internet Directory-Enabled Applications. http://www.ietf.org/rfc/rfc2377.txt, September 1998.

[12] Erik Guttman et al. Service Location Protocol, Version 2. http://www.ietf.org/rfc/rfc2608.txt, June 1999.

[13] HomeRF Working Group, Inc. HomeRF Frequently Asked Questions. http://www.homerf.org/learning_center/faq.html, 2000-2001.

[14] IBM. *SMAPI Developer's Guide, IBM ViaVoice SDK for Linux*, March 1999.

[15] IBM. IBM Speech for Java. http://alphaworks.ibm.com/tech/speech/, 2000.

[16] JDOM Project. JDOM XML API. http://www.jdom.org/, 2000.

[17] Steve Kille et al. Using Domains in LDAP/X.500 Distinguished Names. http://www.ietf.org/rfc/rfc2247.txt, January 1998.

[18] Paul Kulchenko. SOAP::Lite Perl Module. http://www.soaplite.com/, 2001.

[19] Michael L. Dertouzos. The Future of Computing. *Scientific American*, August 1999.

[20] Object Management Group. *Common Object Request Broker Architecture, v2.5*, September 2001. CORBA Overview, Chapter 2.

[21] Steven G. Pennington. Architecture for an Ambient Computing System. Master's thesis, University of Kansas, July 2000.

[22] Rabbit Semiconductor. *RabbitCore RCM2100 User's Manual.* Z-World Inc., 2001. Part no. 019-0091.

[23] Daniel Ridge et al. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. *IEEE Aerospace Proceedings*, 1997.

[24] Sun Microsystems, Inc. Java 2 Platform, Standard Edition. http://java.sun.com/j2se/, 1995-2001.

[25] Sun Microsystems, Inc. Java Naming and Directory Interface (JNDI). http://java.sun.com/products/jndi/, 1995-2001.

[26] Sun Microsystems, Inc. JDBC Data Access API. http://java.sun.com/products/jdbc/, 1995-2001.

[27] Sun Microsystems, Inc. JINI Network Technology. http://java.sun.com/jini/overview/, 1995-2001.

[28] Carnegie Mellon University. CMU Sphinx: Open Source Speech Recognition. http://www.speech.cs.cmu.edu/sphinx/, 2001.

[29] Mark Wahl, Tim Howes, Steve Kille, et al. Lightweight Directory Access Protocol (v3). http://www.ietf.org/rfc/rfc2251.txt, December 1997.

[30] X10 Wireless Technology, Inc. X10 Transmission Theory. http://www.x10.com/support/technology1.htm, 1997-2001.