SECURITY MODEL IN THE AMBIENT COMPUTATIONAL ENVIRONMENT

By

James Mauro

B.S.Computer Engineering
University of Kansas, 2002

Submitted to the Department of Electrical Engineering and Computer Science

and the Faculty of the Graduate School of the University of Kansas

in partial fufillment of the requirements for the degree of

Master of Science

Dr. Gary Minden(Committee Chair)

Dr. Arvin Agah(Committee Member)

Dr. Perry Alexander(Committee Member)

Date of Acceptance

Acknowledgments

I'd like to thank the following people for making this work possible:

- Gary Minden for supporting my work on ACE, advising me on how best to proceed and allowing me to work on ACE for the past five years.
- Arvin Agah and Perry Alexander for being on my committee and making my time at KU enjoyable.
- Leon Searl for teaching me good software practices and helping focus my work on ACE.
- Eric Akers, Renzo Hayashi and Olaf Landsiedel who all contributed to the current version of ACE, both in ideas and in the written software.
- My Parents, James and Kathy, for supporting my time as a student.
- DARPA, NFS, Sprint and the University of Kansas for funding my work on ACE.

Abstract

The Ambient Computational Environment (ACE) builds integrated rooms where services are integrated throughout the building. The services control the lights, projectors, cameras, etc all of which can be controlled by the users through their mobile desktops and with other non-traditional means such as talking to services. The following thesis describes the construction and security of the atomic objects within the environment. These atomic objects are called services in the architecture. The thesis describes communication mechanism that the services use, a secure form of the RMI protocol. It also discuses the use of the Transport Layer Security (TLS) for authorization, the use of TLS and Advanced Encryption Standard (AES) for data encryption and the use of Keynote Trust-Management system for distributing permissions to the services.

Contents

List of Figures

Chapter 1
Introduction

Even though computers have become more powerful and more ubiquitous, the modes of accessing their processing power have not significantly changed since the introduction of the keyboard, monitor and mouse. Programs and sessions are designed to run on one machine at a time. They are tied to that machine is such a manner that upgrades, maintenance, and crashes causes both the programs and sessions to terminate and return to their initial states on restart. The Ambient Computational Environment (ACE) attempts to create a system that allows users to move from room to room and take their sessions with them. ACE attempts to complete these goals by building simple independent services that can be used to control the environment. More complex services can be created by federating the simple services into more complex services by way of managers. The subject of this thesis is the composition of these services and the methods they use to communicate and secure their transactions.

The rest of this chapter describes the reasons for the work. Chapter 2 describes work related to the subject of this thesis. Chapter 3 describes the ACE environment and the requirements placed on the designed services. Chapter 4 describes the middleware used to implement the services. Chapter 5 describes the methods used to secure the communications and authenticate the users. Chapter 6 describes the method for authorizing users to perform actions within the environment. The Chapter 7 describes possible extensions to the environment and the services within. Finally, the chapter 8 describes what improvements, additions, and accomplishments have been made.

1.1 Motivations

As more and more devices are added to a room it becomes more difficult to integrate them into a cohesive unit. The difficulties include:

- GUI Interfaces - Devices usually ship with a standard GUI interface, but these interfaces are usually limited. They won't work with competitors devices and may not work with all machine types or OSes.
- API Interfaces - Most devices do not ship with a programmer accessible way to access the device either through library access or directly access the hardware. Those that do provide interfaces the API is not consistent between vendors or even models.
- Network Access - Network access is either unavailable or limited to a web interface.
- Security Model - The security model, if one even exists, is limited to an all access or no access approach.

The purpose of this thesis is to propose a system where some of these short comings can be addressed. The thesis describes a framework where new devices can be introduced into the system using the current GUI's, even though newer features of a device might not be available to the older software. The thesis also describes a framework that could be extended to permit non-traditional interfaces like voice and motion recognition and novel identification methods.

Chapter 2
Related Work

In this chapter, work that encompasses the similar problems to those that the thesis covers are addressed. However, ACE goes beyond these systems by including security (both authentication and authorization) and a well defined interface to access the services. The work includes the Remote Method Invocation (RMI) system , the JINI architecture, and the Ninja architecture.

2.1 Java Remote Method Invocation

Remote Method Invocation (RMI)[17] is a method where by Java programs can hold a reference to an object that is ultimately stored in another instance of the Java Virtual Machine (JVM). The local JVM can access the remote object as it would access any other local object. The RMI system provides a number of utilities for creating the objects and registering them on the network. Unfortunately the RMI model suffers from a number of issues.

RMI allows for the remote methods and fields of an object to be accessed remotely. The object provides a stub for the remote clients that acts as a proxy for the remote object. The stub takes method calls from the clients, packages the call and the arguments and transmits them to the actual object. The remote object then executes the method and returns the results, either the return value or the exception. The execution of the method call occurs in a stateless manner and call parameters are not cached by the remote object. The actual implementation of the stub is created using the `rmic` program, which takes the implemented remote object and generates a stub that implements the same interface as the remote object. When a remote object is initialized, the remote object registers its self with a service called `rmiregistry`. The `rmiregistry` stores a serialized copy of the stub under an unique name that other client services can use to get a copy of the stub.

A RMI object can differentiate between hosts the users are connecting from, but cannot differentiate between users. This means that the different users on the same host would have the same permissions for calling the methods. It security model allows blocking of specific hosts and ports, but does not allow per user security. An object can determine the host of the client, but it cannot determine the user without the explicit transmission of the user credentials in each method call[18]. The RMI model also suffers from the client having no direct way to verify the integrity of the remote object before the call is made.

In ACE, the communication system that is used is based on the same interface that RMI uses. An Enhanced ACE RMI system would appear to be a Java client to be no different than an normal RMI system. The Enhanced RMI system would have a greater capability to provide for fine grained permissions on executing specific methods under specific conditions.

2.2 Java JINI

Sun's JINI[16] technology builds upon their RMI work to build a system where clients can discover and use services distributed throughout the network. It attempts to make resources on a number of computers appear to be running on a single unified system. The JINI system defines a discovery and join protocol for the network and provides a mechanism for clients to look up services within the network. The JINI however provides no security technology on top of the normal RMI security apparatus.

The JINI system of services provides a way for services to find each other on the network. The system accomplishes this task through three protocols called Discovery, Join, and Lookup. The protocols allow for clients to locate services in the network and provides some differentiation between them. The Discovery protocol is used by both the clients and the services to locate the Lookup Service. During the Join protocol a service uploads its information (class, service location, attributes, etc) to the Lookup

Service, which agrees to save the information for a specified amount of time. The agreement to save the information for the specified time is called a "lease". Finally when a client wants to use a service, it uses the Discovery protocol to find the Lookup Service then uses the Lookup Protocol to request services that match the interface and attributes one is looking for. After the service is found, the client contacts the service directly over the RMI protocol.

Due to the fact that JINI is built on top of RMI, the security for the user is bound up in the security for the RMI model. It provides no additional security features that do not exist in the RMI model.

While JINI provides a lookup service, the lookup service it provides does not different ate between the physical locations of the services. The ACE's Service Directory can search both by type and by physical location. The JINI style of services also have no defined interface for accessing any of the services except the Lookup Service and the services that are provided by default are not arranged in a manner that would be conducive to having an unified method of accessing all the services regardless of type.

2.3 Ninja

The Ninja project attempts to build a network of services that are scalable, fault-tolerant, and highly-available. It has built a number of core services for lookup and discovery. It has implemented an RMI called NinjaRMI [20]. The Ninja Architecture provides a number of services to facilitate building the network.

The NinjaRMI is similar to the Sun's version of RMI. It's major difference is that it allows both the client and the server to examine the address and ports of the other side. It is also designed to allow for different transport channels for remote object communication, including Multicast and UDP instead of the regular TCP connections. It was designed against the feature set of the RMI available with the 1.1 version of the JDK.

In order for services to find each other on the network, Ninja builds a set of hierarchical service locators called Secure Discovery Services (SDS)[3]. The SDS's attempts to resolve the query and if it cannot resolve it, it sends the request onto the higher levels. The system conducts it's queries using XML messages, but the matching is constrained to an exact match for the service requested.

The ACE project is similar to the Ninja project in many respects, but the limitations of the Ninja model made it preferable to implement a new model. For instance, the NinjaRMI allows for the replacement of regular TCP sockets with TCP sockets secured over the TLS protocol, but the model does not allow for information about that session (like who the user making the method call) to be passed to the calling user. It also implements no authorization mechanisms except those mechanisms that are inherent to RMI. The SDS-style lookup services does not allow for services to be searched by subclasses only directly by type and location which would limit arranging services in a hierarchy to take advantage of the object oriented paradigm.

Chapter 3
ACE Overview

The Ambient Computational Environment (ACE) project was tasked to set out and build a pervasive system that enables users to have long lived workspaces and mobility within the environment, regardless of the room or the machine they are currently using. This allows for a user to move within the environment with his or her workspaces following them around.

The following two scenarios are examples of what the ACE framework hopes to accomplish:

Herman walks into a conference room with his Personal Interface Device (PID). The PID identifies him to the conference room as "Herman". He requests access to the devices in the room and he is granted permission to access the services in the conference room. He the proceeds to set up the conference room by setting up lighting levels, moving his presentation to he main projector, connecting a remote site's video feed to the room's main monitor, and so forth as he sets up for the conference. He then notes the current set up in his PID and leaves. Some time later, when the meeting is scheduled, Herman enters the room, requests access to the resources once again and recalls the noted setup, restoring the room to the configuration he set up earlier. During the meeting, Herman can direct the room without the use of the PID, but by using gestures and voice commands. For instance, he can direct the cameras to look at a certain seat by vocalizing the command to the camera to "look here" and pointing to the seat in question.

Holly checks her computers and sees that her working context has be running continuously for 497 days before she leaves for the weekend. During the weekend the office support staff replaces her computer with a new upgraded one. The staff person unplugs the old computers (without saving Holly's work or shutting down the old computer). When Holly returns she identifies herself to the new machine and checks the age of her context. The context has been running continuously for 499 days and the programs left running before the weekend are still running.

The ACE concept believes that computing environments contain the following features[11]:

- Computational resources are readily available throughout the space in which people move. The term“Computational resources” includes CPU cycles, memory, storage, display, wired and wireless communications, sound input and output, video input and output, i.e. anything connected to computing.
- Users co-opt computational resources in their vicinity for their use.
- Computational sessions are long-lived and mobile beyond the extant for their individual machines or instantiation.
- The computational environment re-acts to user voice commands, gestures, and computer commands and maintains and individual model of how the specific users act.

3.1 Architecture

In order to implement the above scenarios like Herman's and Holly's experiences, an architecture for ACE has been developed. The services comprise the atomic levels computational resources. The services can be arranged into larger federations of services to perform larger, more complex tasks. A number of services have been created to facilitate the creation of these federations. Finally, the environment contains information about were the hardware services are located in the environment (like camera mounting points).

3.1.1 Service Architecture

The services within the ACE environment are designed to be simple atomic elements that perform only one function. If a service captures video, it would not know how to display or convert the video stream (those functions would be handled by other specialized services) and if a hardware device performs more than one type of operation it is separated into multiple services. Long-lived services form temporary federations in order to create higher level services like end to end video.

Services communicate with each other and clients in two distinct ways, the Control Channel and the Media Channel. The Control Channel is a reliable in-order channel that provides the main communication mechanism for the services. The Control Channel provides a way for the services and clients to communicate messages for controlling the service. The Media Channel is for those services where reliability and in-orderness is not as important as timeliness. The services that the Media Channel are designed for are the audio and the video services.

Take for instance the setup of service for a Pan-Tilt-Zoom (PTZ) camera in Figure 3.1. The camera would create two services to handle its functions. The first service contains the functionality to handle the mechanics of the camera (the pan, tilt, and zoom functions). The second service would handle the camera's video input stream. Both of the services communicate with the controlling clients via a Control Channel. The video service transmits its video to other media services to display or process the video via the Media Channel.

3.1.2 Service Federations

While single function services are interesting, they are not particularly useful. In order to perform higher level functions a video conference between two site, the services must be grouped into larger federations. These federations are managed by a client, called a manager. A manager searches for services on the network that can perform the required tasks and then configure those services to work together. The

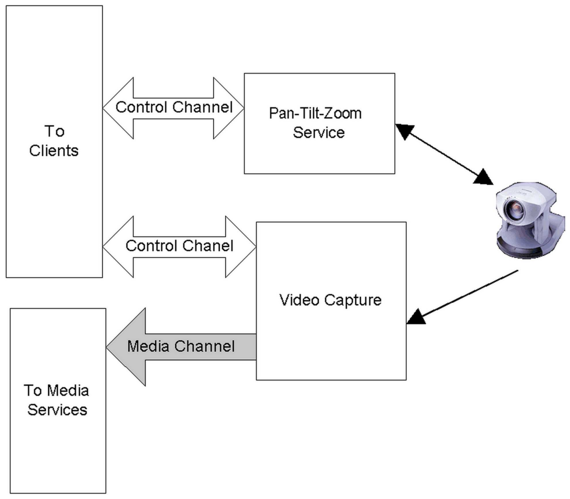


Figure 3.1: A view of a single device and the services it creates, as well as connections to other clients and services.

manager handles all information transfer and coordination between the services except for the data that is transmitted in relationship to the audio and video streams. The federation stays intact only while the manager controls the services. After the federation has out lived its usefulness, the services can be reconstituted into a new federation with different members.

Figure 3.2 shows an example of a federation to create a one direction tele-conference. The conference needs audio and video to be transmitted from one site to another site. The manager finds an audio capture and a video capture service at the remote site and an audio player and video player at the local site. Since the traffic needs to be compressed before it is transmitted over the network, the manager finds two converters to compress, uncompress, and convert the data from the capture format to the display format.

3.1.3 Core Services

In order to facilitate services working together a number of "core services" are required. The core services enable the services to find each other, look up users, discover the resources of the rooms and to retrieve certificates for user authorization. In

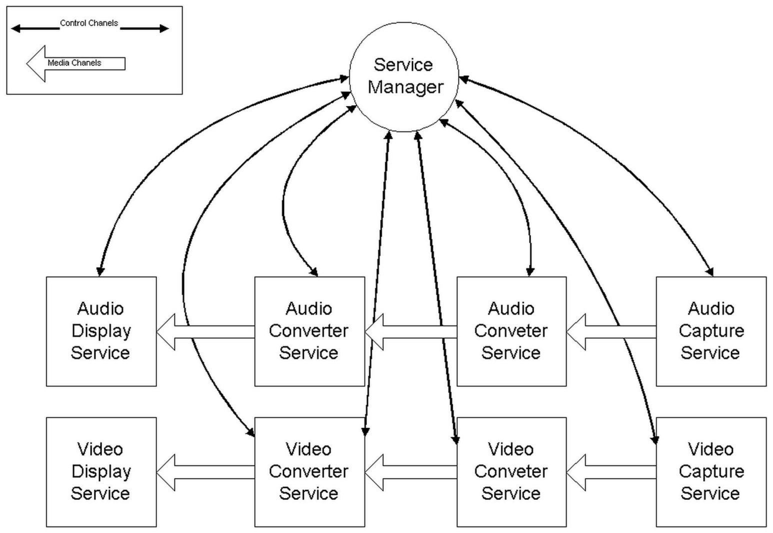


Figure 3.2: A view of a federation of services designed to create an end-to-end video conference service.

keeping with the idea of simple services for ACE, four separate services were created to accomplish the tasks. The four services are called the Service Directory, the User Database, the Room Database, and the Authorization Database.

Figure 3.3 shows how the services use the core service. The service would register its self with the Service Directory to allow others to connect to it and use the service directory to find other services (Line 1). Next the service would communicate with the Room Database to discover which room it is located in and use that information to configure itself (Line 3). A client would discover a service's current address by connecting to the Service Directory and querying it (Line 5). A client would then connect to the service (Line 6) and attempt to gain authorization. To verify the authorization the service would contact the User Database (Line 2) and the Authorization Database (Line 4).

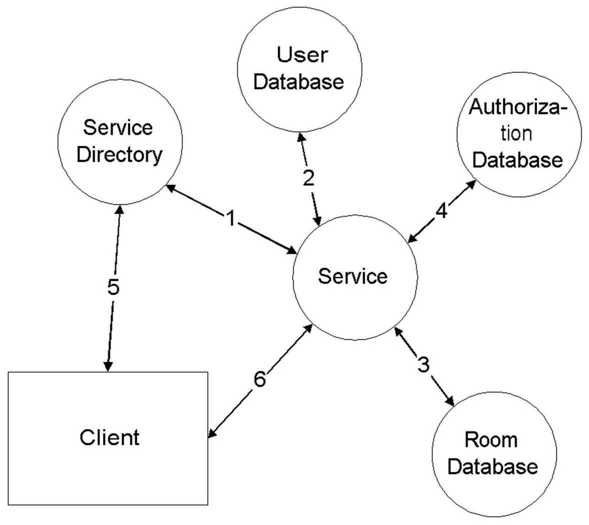


Figure 3.3: The figure shows the relationships between the core services, a service utilizing the services and ultimately the clients that would want to use the service.

3.1.3.1 Service Directory

The Service Directory is the core service that handles registration and unregistration of the services and allows for clients to lookup the registered information. Since the Service Directory is used to locate all the other services, it is the only service within ACE a fixed address.

The Service directory keeps track of a service for a specified amount of time, called the lease time. A service registers its address, name, location, and service class with the Service Directory and the information is stored for the specified lease time. The service recontacts Service Directory before the lease time expires to reset the timer. If a service does not renew its lease before the lease expires, the service is removed from the searchable list of services.

A user searches for a service based on one or more of the following characteristics:

- Name

- Room
- Machine
- Service Class (including sub-classes)

3.1.3.2 User Database

The User Database contains all the information about users in the system. The database indexes this information by the public key of the user. Information about the user includes:

- Public Key
- Name
- Login Name
- Login Characteristics

The login characteristics includes information identifying users for login types. These logins includes information like passwords, finger print scanner, and iButton id's that are used to identify the users when gaining access to the system. The user's login information is not accessible to normal users, but all other information is accessible.

3.1.3.3 Room Database

The Room Database contains information about the rooms set up inside of the ACE environment. The Room Database contains information about the room's size and locations of objects in the room. It also contains information about what machines are nominally located in the room, what capabilities they have and what services they are required to run due to hardware configurations. All information stored in the database is available to any user, but editing the information is restricted to specified administrators.

3.1.3.4 Authentication Database

The final core service is the Authentication Database. It stores Keynote assertion certificates [2] that specifies permissions used within the system. These certificates specify which users are allowed to access services. The services are indexed by the public key of the user, room, and time the certificates are issued to cover. The services query the authentication database while determining the correct permissions of the user. While everyone is able to read the database contents, only administrators are allowed to add or update information.

3.1.4 Room Architecture

Finally, information about were services are located are stored inside of the room database allows for services to be associated with a room. The room is the largest unit inside of the ACE. Figure 3.4 shows one such conference room. Services located inside of specific rooms tend to be those services with a some required physical connection to the room (for instance they utilize hardware mounted in the room). Services with no specific room (like media converters) can be moved from room to room at the will of the system and are not included in the description of any specific room.

The room contains two PTZ camera services, a projector, two microphones, four speakers, a general purpose computer, and a number of iButton[4] authentication devices. These access to these devices are granted when someone enters and identifies themselves to the room. The users would have instances of a manager that could the utilize the services to perform higher functions. For instance a user could use a microphone and a video input service from a camera to set up a service that would listen for commands and then determine what the command was referring to by looking at the actions of the user through the video camera.

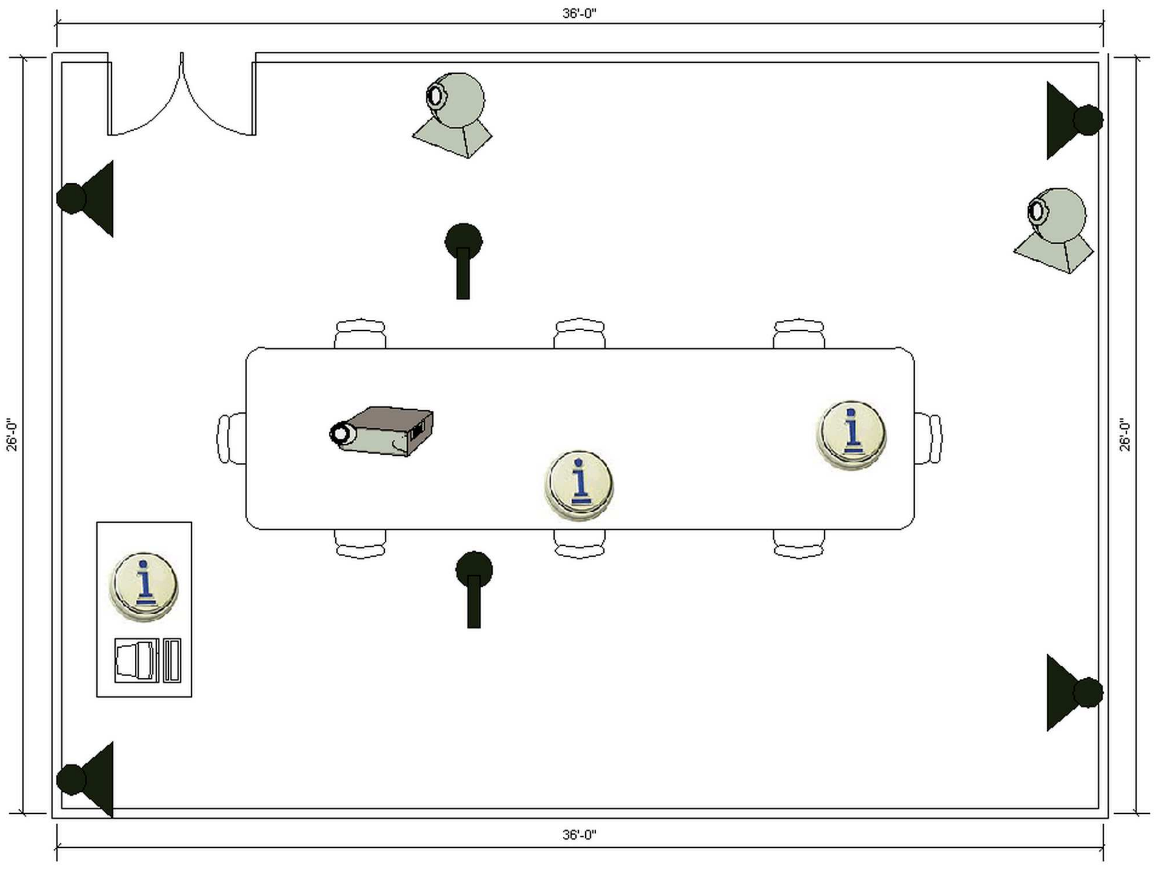


Figure 3.4: A view of a possible room. The room contains projector, camera, audio capture, audio replay, CPU and iButton authentication devices[4]

3.2 Implementation

In order to build the required architecture, the environment was implemented with a number of features. The first feature is the Service Hierarchy, which arranges services so similar services can be used by just knowing their common methods. Secondly, an Enhanced form of RMI was created in order to allow for the required encryption, authentication and authorization for the services to be implemented. The Enhanced RMI generates classes, called stubs, that act as proxies for the services on the remote hosts. The stubs and services implement the actual interface for the service. See Appendix B for an example and source code of an implemented service.

3.2.1 Service Hierarchy

In order to ease the implementation of services and to simplify bringing new services on line the Service Hierarchy was created. The intent of the Service Hierarchy is to group together services in the environment in such a way that all the services that share similar functionality share as much of the same interface and implementation as possible. This allows for older clients to utilize newer services by referencing the lower levels of the interface. The interfaces are arranged into a tree with the most basic functions at the top of the tree and the leafs of the tree being the most specific.

For example, the Epson 7350 series of projectors have a picture-in-picture function. The projector's functions of changing the input for the projector is defined by the **Projector** interface, while the picture-in-picture system is accessed by using the **Epson7350Projector** interface. This allows for a client that just knows on how to use a projector to use the Epson 7350 by just accessing the Projector interface and ignore the more advanced features of the Epson7350Projector interface.

A number of interfaces have been defined in the ACE to implement the most common services within the environment. Figure 3.5 shows an example of part of the tree. The base of the tree contains the Base and Service levels that define the basic properties of any service. All Services should extend off the service level, since the base level provides no methods (it exists for possible future expansion). The next level contains the main branches of the Hierarchy, Device, Media, and Databases which define the three main types of services. The Device branch represent hardware devices like cameras, projectors, iButtons security devices, etc. The Media branch handles devices like audio and video capture, display and streaming. The Database branch that contains data stores like the Service Directory and Authorization. Some services, like the Host App Launcher, do not fit into either of the three main categories and extend directly off of the service level.

Directly under the Database level consists of a implementation only level called SQLDatabase that is not exposed to the clients, but is used to group together the SQL

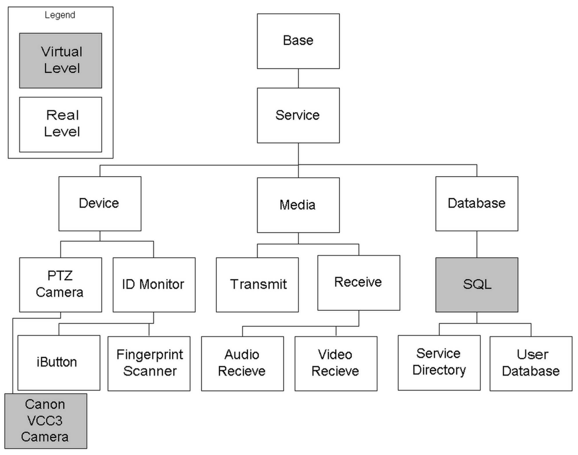


Figure 3.5: The following is an example of the hierarchy. The blue items are members of the hierarchy that are visible to the clients. Green members are implementation only levels.

functions that are used by a number of sub-services. These levels are referred to as virtual levels since they have no corresponding interface and therefor from the clients point of view do not exist.

For more information about what is contained within the tree look to Appendix A for a complete list of the implemented services.

3.2.2 Enhanced RMI

In order to communicate between services, an enhanced form of the Remote Method Invocation (RMI) was created. The Enhanced RMI system is a stateful version a remote procedure call (RPC) system where remote services appears as local objects that can be used and manipulated like any other object. In reality the local object is a proxy for the real object. These proxies, also known as stubs, are generated from the implementation of the service. The stubs have the same interface as the service, but implement a communication system to the remote object instead of the actual object.

3.2.3 Security and Authentication

In order to provide the requested services a number of security policies and processes must be developed. These policies include defining how to representative users within the system, how the users are authenticated and how the services are secured.

3.2.3.1 User Identification

Users within the environment need a unique identification to differentiate them from other users. The most appropriate way to identify a user would be his or her public key. Public keys also have a number of other properties that are useful in authenticating users and passing permissions between users. It allows users on separate ACE environments top have an identification token that could be used in other environments. It also allows for user authentication and authorization by the services without transferring a known secret between them.

It does have a drawback, the ability of the users to refer to each other would be difficult. The problem is mitigated due to the presence of User Database that is used to map user's real names and login names to their public keys. A user could use a shortened login name and on the fly the users pubic key is retrieved from the User Database, similar to how Unix uses the passwd file to map user login name to the user's identification number (UID).

3.2.3.2 Remote Authentication and Trust Management

Since the ACE uses network services as building blocks to perform actions, a system needs to be developed to both authenticate the users and a system to define what users do within the individual services. Since users are identified by their public keys, the private parts of the keys is used to authenticate the users. The ACE uses the mechanisms embedded in the Transport Layer Security (TLS) protocol [5]to perform this authentication. After a user has been authenticated, the levels, processes, and which he or she effects needs to be determined. The exact permissions that each user has needs to be shared and transmitted between each of the services. For this purpose the ACE uses the Keynote Trust-Management system [2].

The ACE grants and denies of access on a number of conditions including:

- User ID
- Service being accessed
- Current time
- Room the service is located inside
- Method being accessed on the service

Other conditions are possible but dependent on the implementation of the service using them.

Authorization is granted by the Keynote compliance checker based on the assertions and conditions it contains. Due to the nature of the Keynote Trust-Management System authorization can be passed between users by creating new assertions and passing them to the service.

3.2.3.3 Data Encryption

The ACE environment requires that data being transmitted be protected. The reasons for the protection are to prevent unauthorized individuals from accessing in appropriate information and to prevent sessions from being replayed in the future to repeat an action. To handle these functions on the Control Channel, the Transport Layer Security (TLS) protocol is used. For the Media Channel a system of symmetric encryption using the Advanced Encryption System (AES) [12] is used. The keys for the Media Channel are shared using the Control Channels.

Chapter 4
Enhanced RMI

The standard Java model for objects communicating across the network is to use the Remote Method Invocation (RMI) system. While this method works well for abstracting away the problems of creating remote interfaces and the language constructs needed for services to communicate, the implementation of the communication protocol makes adding per user and per method security difficult. Due to the security concerns, the ACE project used an enhanced form of the RMI protocol.

The new Enhanced RMI system implements the same interface as the standard RMI system. This allows the Enhanced RMI system to use the same tools as the standard RMI and to let the objects have interoperability when needed. The modifications made for this these to the RMI model included a reimplementation of the Unicast Remote Server to provide the spadeful, per-user security required for ACE.

4.1 Standard RMI

The process of implementing a standard RMI service consist of implementing two separate pieces of software, the first part consist of the interface for the RMI service. The interface indicates to the JVM that it is remote by extending the Java interface **java.rmi.Remote**. The second part consists of the implementation of the service. The implementation uses the class **java.rmi.server.UnicastRemoteServer** to access the mechanisms of RMI by either extending the class or using the **exportObject()** method to export the object. The stub classes used to access the **java.rmi.server.UnicastRemoteObject** remotely are generated using the *rmic* program provided by the standard Java Development Kit.

4.1.1 Standard Security Model

The basic security model for RMI provides very little extra in the way of security. It differentiates between hosts and can ban hosts from connecting to the object, but it cannot however differentiate different users on the same host. It does not provide any mechanism for securing the data in transit. There exists a way to add custom sockets which could use a mechanism like TLS allowing for per user identification, but this method does not allow the user's ID to be passed back to the method being called. The end result is that the TLS socket will allow valid system users to be let in, but no other properties could be judged on whether the user could access the module.

The most common solution around this problem is to pass credentials as an arguments to every method call. While this solution accomplishes the goal of allowing a program to have per user access, it forces the developers to include code in every method to handle the security verification, cause the execution of a method to begin before security has been checked (possibly tying up unneeded resources), as well as dirtying the method signatures with extra, unneeded arguments.

4.2 Secure Unicast Remote Server

As a solution around the deficiencies presented by the standard RMI model a new remote object was implemented under Java. This object provides the mechanism that allow the security system required by the ACE environment to be implemented. The resulting class, called **edu.ku.ittc.ACE.Server.SecureUnicastRemoteServer**, implements a stateful server that is used to both identify and check his or her permission before the method is executed. It simplifies the work a programmer must do to use the ACE security model and localizes any changes that need to be made when security updates occur to just one classes for all of ACE.

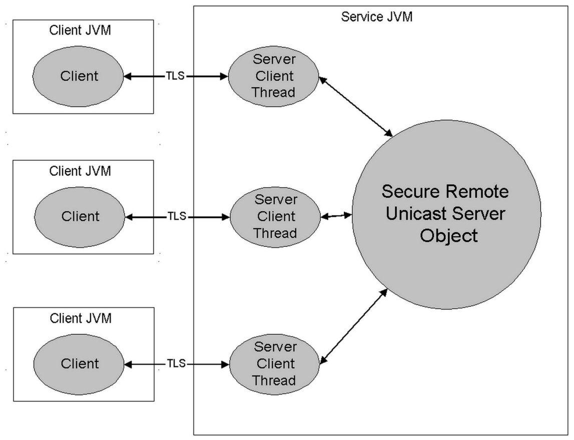


Figure 4.1: The graphic represents the execution environment of the edu.ku.ittc.ACE.Server.SecureUnicastServer. Each client would have its own thread in its own JVM, while the sever has one client thread per client. The service its self may have one or more threads.

4.2.1 Basic Architecture

The **edu.ku.ittc.ACE.Server.SecureUnicastServer** object was designed to implement the **java.rmi.Remote** interface. As such, it uses most of the RMI programs (like *rmiregistry*) without any modifications to the programs. The helper program, called *StubGenerator*, generates stubs for the in a similar way to the *rmic* for standard RMI.

4.2.1.1 Secure Unicast Server

The Secure Unicast Server is composed of two object classes classes, **SecureUnicastServer** and **ServerClientThread**, both classes are members of the **edu.ku.ittc.ACE.Server** package. Each class has its own thread of execution that handles a different part of the communication system. The **SecureUnicastServer** thread handles new incoming connections and starts a new **ServerClientThread** for each connection. While the class **ServerClientThread** handles the individual requests and the state of each client. There is one **ServerClientThread** per client that connects. Figure 4.1

shows the relationship between the two server classes.

A remote object uses the **SecureUnicastServer** object by either extending the class or implementing a remote interface and calling the **exportObject()** method. Upon initialization of the extended object or the calling of the **exportObject()** method, a thread is started to handle the incoming connections to the server. When a client connects, the **SecureUnicastServer** temporarily accepts the connection and creates an instance of **ServerClientThread** to handle requests. The **ServerClientThread** then authenticates the user. When a request to execute a remote method comes in, the **ServerClientThread** runs the required per method authorization and then executes of the method. If the execution is allowed, the method is executed in the thread controlled by the associated **ServerClientThread**. The result of the execution is then returned to the client.

4.2.1.2 Stub Generation

A stub generated for each **SecureUnicastServer** that implements the communication mechanisms for connecting to the remote object. When a remote object registers itself with the *rmiregistry* or when the object is serialized to an output stream, the stub is transmitted instead of the remote object. This stub contains the information necessary for the service to be contacted and its methods to be accessed.

The stub generator works by loading the compiled interface of the requested class. The class then is tested if is a remote interface. The generator then writes out a new Java source file containing the proxy's implementation. The generated class implements the same remote interfaces as the requested class. The resulting stub is then compiled using *javac*. After compilation a service creates instances of the stub class to pass to those clients who are interested in connecting to the service.

4.2.1.3 Messages Between the Stubs and the Server

When a stub wants to execute a method on the remote object, a message object is created. The message object, contains the method definition and either the arguments to the method, the return value or the generated exception to the method call. The method definition contains the name arguments and return types of the method as well as any declared thrown arguments. The format for the method declaration is the same format as the method signatures in the Java Language Specification [8].

When calling the method, the client packs the method signature and arguments via serialization and sends the object to the remote object. The sending thread on the client blocks until the response is received.

Upon receiving a message object, the remote server determines the **java.lang.reflect.Method** that the method signature in the message belongs to. The method is then executed in the receiving thread and the results are gathered. A return message object that contains the same method declaration as requesting object and a return value or exception is generated depending on the execution. The result is the returned to the client over the same channel.

When the client receives the response and unpacks it. The client then takes the return value and returns it to the calling method. If the remote object incurred an exception the returned exception is thrown instead.

Chapter 5
Encrypted Communications

In order to provide both authentication and transport security for the ACE services, the Transport Layer Security Protocol (TLS) [5] was used on the Control Channel. In order to protect the the Media Datagrams the AES [12] and the SHA-1 algorithm [6] is used to protect each packet.

5.1 Control Channel Security

This section describes the use of the TLS protocol in the services. The following sections discuss how keys are managed and how authentication and encryption are used by the services.

5.1.1 Key Management

The individuals in the ACE are identified by their public keys. The public keys are used for two separate functions. The keys are generated by the users and added to the ACE Certificate Authority for use by the TLS protocol and inside of the Keynote Trust-Management System. The users can then utilize their key pairs to identify themselves, to protect their communications and to receive authorization on the system.

5.1.1.1 Key Issuing and Processing

The users in the ACE are identified the pubic part of the asymmetric key. The key is either an RSA or DSA asymmetric key. These keys are formatted into x509 certificates [7] for the use inside of TLS and inside of Keynote Trust-Management System. The keys are also signed by a Certificate Authority (CA) to verify the validity of the key.

The initial public and private key pair is generated when the user account is created. The private part of the key is restricted to the user, while the public part is placed into a certificate request. The request is then signed by the ACE certificate authority to verify the validity of the key. The resulting signed key is passed back to the user with the CA's certificate. The user can then be added to the User Database. The when the signed key is uploaded to the user database it is converted into the format Keynote uses (a base-64 encoded string of the certificate with the type, in this case x509-base64:, perpended to the the beginning.)

The CA serve the purpose of verifying the key is valid and allowing the keys to be verified between domains. The every valid key is signed by the CA and those users who are no longer allowed are added to the Certificate Revocation List (CRL) that is managed by the CA. Since each ACE domain would have its own CA, but users could keep their original key. This can either be accomplished by signing the remote CA's key or by signing the new user's key by the local CA.

5.1.1.2 User Keys Utilization

Users within ACE need to protect all their private keys from unauthorized access. The private keys are best protected using operating system mechanism. The private keys can then only be read by application running as the user. The user can access his or her private keys for negotiating TLS sessions and signing new Keynote assertions.

Because the keys are used to access the authenticate the users, access to them must be limited. The private portion of the key is stored in a file that only the user can access. These keys can only be accessed after a user has been logged into the system. The logging in process is handled by services known as ID monitors. The ID Monitors are configured to spawn processes off as the user logging in. These process are usually managers that allow rooms to be configured or sessions to be started. These processes then access remote services using the users public/private key pair.

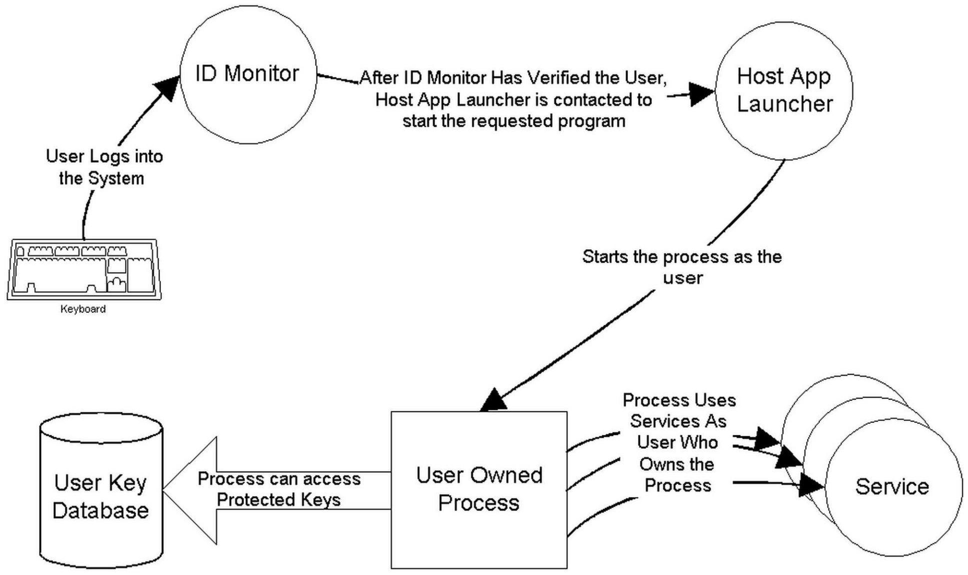

ID Monitor
After ID Monitor Has Verified the User, Host App Launcher is contacted to start the requested program
Host App Launcher
User Logs into the System
Keyboard
Starts the process as the user
Process Uses Services As User Who Owns the Process
Service
User Key Database
Process can access Protected Keys
User Owned Process

Figure 5.1: The login process using the ID Monitors

5.1.2 Authentication and Encryption using TLS

The Transport Layer Security protocol provides both authentication for accessing the service and encryption for the conversations that occur on the Control Channel. It creates a secure authenticated channel on TCP-like data streams.

Authentication in the TLS protocol uses a two-round handshake to authenticate the users and negotiate the session key and algorithm used. The protocol uses x509 certificates [7] to identify users. The x509 certificates contain information about the user and the public portion of their asymmetric key. The certificate is also signed be a trusted third party to verify the authenticity of the key and avoid the man-in-the-middle style of attack. After the session is negotiated the data is compressed and encrypted using a stream cipher like RC4 or a symmetric block cipher like AES or triple-DES.

5.2 Media Datagram Security

Since the main properties of the Control Channels are unneeded for applications like transmission of audio and video over the network, a second communications sys-

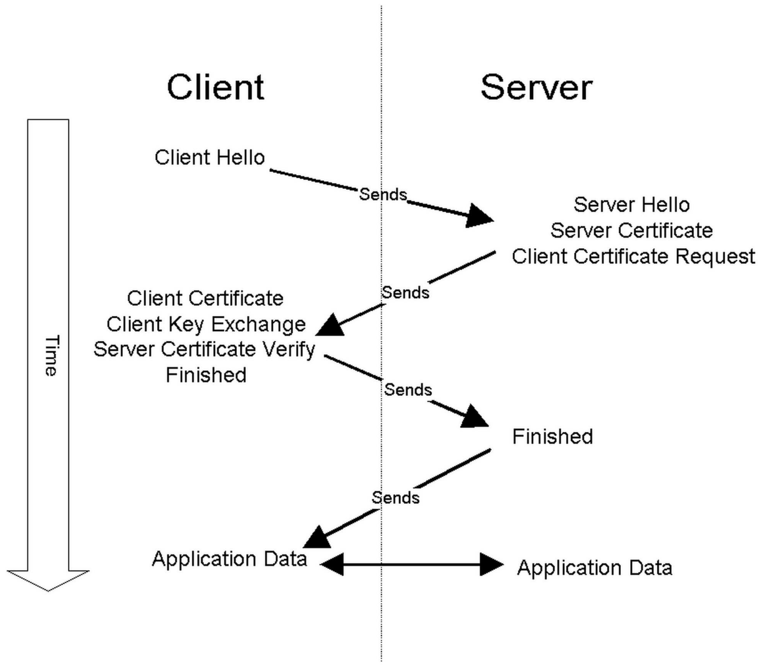


Figure 5.2: The communication process to negotiate a TLS session. The handshake is a two phase session where ID's are exchanged then the session key.

tem was developed to communicate the data. The system uses RTP datagram packets [14] to handle the transmission of the audio and video frames. These frames are then wrapped inside of an encryption frame the protects the data while in transit.

5.2.1 Key Generation

Keys in the ACE are generated by the clients setting up the media services. The key is transmitted out-of-band by the Control channels for the clients and services that want to access the media stream. The key consists of an array of bytes that are used to specified the key and initial vector (IV) for the AES encryption. The key is generated using a random number generator due to no specific requirements for the AES key. The key exchange system is also designed to allow the keys be rotated at specified instances too prevent too much data from being used with one key.

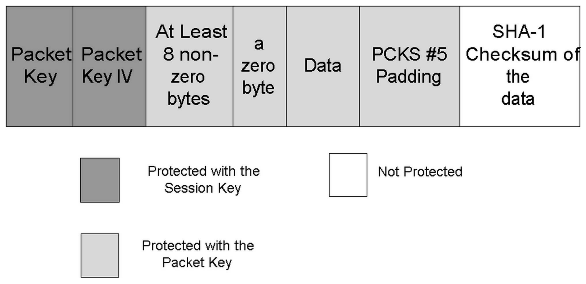

Packet Key
Packet Key IV
At Least 8 non-zero bytes
a zero byte
Data
PCKS #5 Padding
SHA-1 Checksum of the data
Protected with the Session Key
Not Protected
Protected with the Packet Key

Figure 5.3: Packet format for the media packets.

5.2.2 Packet Protection

In order to protect the packet, a media packet format is defined. The format include the header, footer, checksum algorithm and encryption algorithm. The packet is encrypted using the AES algorithm and the SHA-1 hashing algorithm is used to create data integrity.

The packet format takes the data in the packet and the key used to protect that packet. The purpose of the packet format is to prevent common data in the media packets to be used for breaking the key and to detect bit error introduced in transmission. The packet format is showed in figure 5.3. The packet format is based on the PKCS #1 [9] format. The packet starts with a 256-bit header that contains the packet key. The actual data is then headed by at least 8 non-zero bytes to remove any expected data at the beginning. A zero byte is then inserted into the stream to signify the end head padding. The data is placed into the packet in the next block. The data is terminated by PKCS #5 padding[10] to delineate the end of the data and to correct for block size mismatches. The header, data and footer are then encrypted with the packet key. The packet is finally terminated by the 20 bytes representing the SHA-1 checksum of the encrypted portion of the packet.

The encryption used to encode the packet is the AES [12] algorithm. Both the session an the packet use the same algorithm AES is a 128-bit block cipher with key of length 128. In order to protect blocks longer than 128-bits Cipher Block Chaining

(CBC) [13] is used. When the session is set up the IV is required as well as the key. The session IV and key are only used to encrypt the key and IV for the individual packet. The packet key changes from packet to packet. This scheme is used to prevent the session key from being overused.

The hash of both the padding and the data is computed using the SHA-1[6]. The reason is to insure some data integrity in the packet. While the media algorithms, like AAC and MPEG-4 can handle individual bit error in transmission, the encryption used would make packet unreadable from the point where bit bit error occurs. As such, if the error does occur, it does not make sense to attempt to process the packet, but instead to just consider such packet to be lost. Because the checksum occurs outside of the encryption system it is only used to insure integrity during transmission and cannot be used to verify that the packet was not altered by a third party.

Chapter 6
Keynote Trust-Management System in ACE

In order to give access to the users in the ACE the Keynote Trust-Management System[2] chosen. The Keynote Trust-Management System defines a way the permissions are assigned to users and have those permissions re-assigned. To use Keynote in ACE, the system of passing assertions must be defined, the conditions that the ACE uses by default and how it is integrated into the service structure.

6.1 Keynote Functionality

The Keynote Trust-Management System is a system for distributing information about the access levels and the conditions needed to access those levels. It embeds the necessary information into certificates called assertions. Assertions describe the principles of the operation and the actions they can perform. The assertions are evaluated using a compliance checker.

All trust-management systems consist of five main parts[2]:

- A language for describing actions
- A system for identifying principles
- A language for describing which actions the principles can perform
- A method for allowing principles to pass their authorizations to other users
- A system for verifying compliance with the above requirements.

The Keynote Trust-Management System uses a language to both describe the actions and to describe the conditions in which an action is performed. The language uses conditions which are connected by boolean statement to evaluate into the possible actions. For instance `APP_DOMAIN == ‘‘ACE’’ && ( ( time < 23000 ) || ( cost < 4536) ) − > WRITE` would give permission to the principle for the WRITE action if the APP_DOMAIN is “ACE” and either the time is greater than 23000 units or the costs are greater than 4536 units. The language evaluates either numeric or string values against the specified variables .

Principles are listed in the assertions as either authorizers or licensees. Authorizers represent the person attempting to perform the action, while licensees represent who, if anyone, the permission is being transfered to. Keynote describes these principles as strings. It does a straight equality check between the users. Even though principles are identified by any sort of string, they are most often identified by their public key. Using public/private key pairs allow for permissions to be re-assigned and the verification of the reassignment to occur via signing of the assertion.

The compliance checker takes in both the trusted assertions and the untrusted assertions and runs a depth first search for permissions against the current state and the user or users who are trying to authorize the action. It attempts to discover the highest possible permission that are granted to the user and returns that value to the user.

The following is an example of a policy assertion

```
keynote-version: 2
authorizer: POLICY
local-constants: KEY1 = "x509-base64:MIIEZzCCA...LCSG0N2ICh"
licensees: KEY1
conditions: (APP_DOMAIN == "ACE") -> _MAX_TRUST;
```

Since policy assertions are implicitly trusted, they do not have to be signed. This policy give the highest level of trust, specified by the Keynote variable _MAX_TRUST, to the user who holds the licensee’s key.

The following is an example of a credential assertion:

```
keynote-version: 2
authorizer: "x509-base64:MIIEZzCCA9CgAw...LCSG0N2ICh"
```

```
local-constants: KEY1 = "x509-base64:MIIEZnb53...ighfkRT4523k"
licensees: KEY1
conditions: ((APP_DOMAIN == "ACE") &&
(time >= 1082390980610) &&
(time <= 1082390980628)) -> "write";
      ((APP_DOMAIN == "ACE") -> "read";
signature: "sig-rsa-sha1-base64:Nt4+XIP...soP+mgjjTXWA=="
```

It specifies whose permission is begin subsumed in the authorizer field and to whom the permission is being granted to in the licensees field. It lists two possible access levels, one being write and the other being read. The permission granted would be based on what is the current time. The signature is that of the authorizer and used to validate if the authorizer actually gave the permission.

6.2 Keynote in ACE

The description of ACE's use of Keynote comprises two separate parts. The first part covers how the assertions are distributed and how the conditions are used inside of ACE.

6.2.1 Assertion Distribution

Assertions are passed to services in three ways. First policy assertions are available to the services upon start-up. Credential assertions are passed to the service by querying the Assertion Database or by passing the assertion to the service over the command channel.

The policy assertions within ACE setup the core of the permissions for the system. In ACE all permissions is given to a user called "ace", who like "root" user in Unix, does whatever it wants. These user "ace" distributes its permission to any user in the system. The "ace" is consistent throughout an ACE domain (different domains would have a different "ace"). All other permissions must be defined by credential assertions passed to or retrieved by the service.

Most of the assertions in the ACE are stored in the Assertion Database. The Assertion Database is a long term storage system for all the services inside of ACE. It stores credential assertions that are searched based on time, location, service type and users who need authorization. All the stored assertions are retrieved and loaded into the compliance checker to do the actual processing on what the permissions are. The compliance checker also determines if the signatures on the credential assertions are correct. When a user logs into a service, the service queries the Assertion Database to retrieve those assertions that could be used.

Sometimes the assertions needed are not stored in the Assertion Database. For instance if a user is visiting from outside the local ACE domain or if the Assertion Database is unavailable. In cases like these the credential assertions are passed directly to the service using a method inherited from the Service level of the ACE hierarchy called **addAssertion**. The method adds the assertion to the current session if the cryptographic signature are verified against the authorizer.

6.2.2 Keynote in SecureRemoteUnicastObject

The control of the Keynote session occurs in the **SecureRemoteUnicastObject** in the **ServerClientThread**. The **ServerClientThread** holds a Keynote context for each user. After a user connects and is authenticated by the TLS session, a new Keynote session is created for the user. The established session loads the default policies and fetches any relevant credentials from the Authentication Database. Then the conditions are loaded in and the permission level is checked. If access is permitted then the **ServerClientThread** executes the method. A user could then add any assertions he or she feels are necessary by calling the **addAssertion** method. Access is checked before every method call and the session is cleaned up upon the user disconnecting.

If a service wants different conditions than the default ones, it adds them to the user section when the user executes the method. The service also determines the current

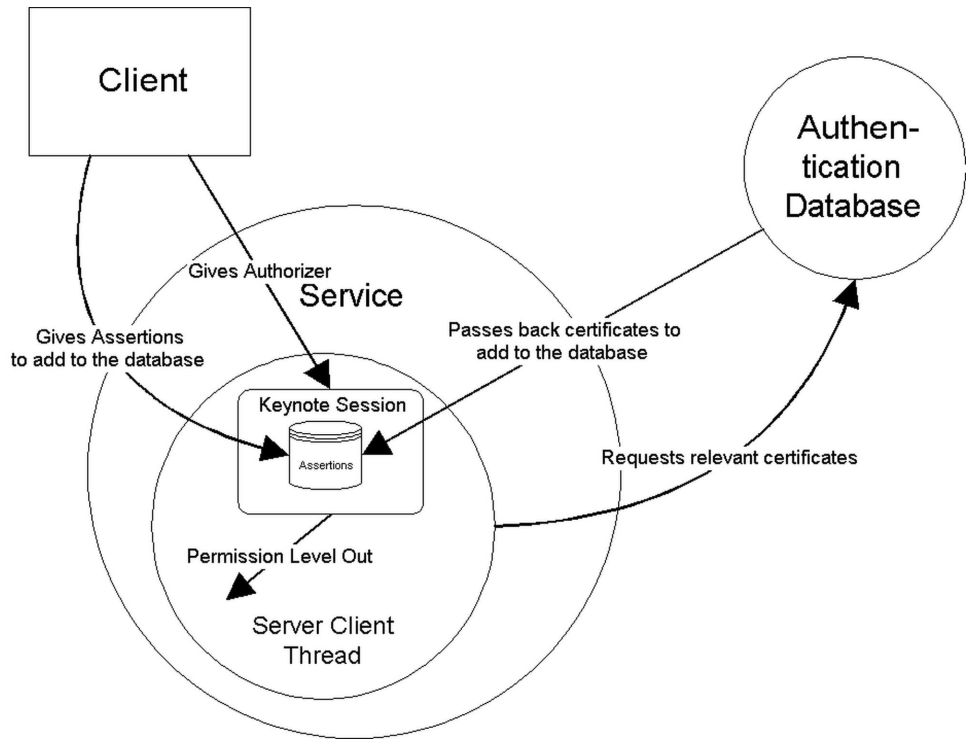


Figure 6.1: Sources of data from the use of Keynote inside of a service

level and then check to see if the new variable raises the permission level as well as getting what the current level is of the user executing the command.

6.2.2.1 Permission Levels

In the ACE system there are four levels of permissions: administrator, write, read and no_access. The permissions are arranged in that order with administrator at the top and no_access at the bottom. A permission level of no_access only lets the method **addAssertion** to be called (and then the number of calls is limited before the system disconnects). Read access levels grants users permission to execute method. Those services that do no want to have users change the internal state without correct permissions should check the user has write permissions before executing the state change. The administration permission gives the administrator full permissions on the system to do what ever they want.

6.2.2.2 Conditions Used

The service architecture enters a small number of environmental variables into the keynote session automatically. The variables include:

- “time” - the current time (in seconds from midnight January 1, 1970)
- “method” - the method being executed
- “service” - the service class
- “room” - the room the service lives in
- “machine” - the machine the service is being executed on

Other conditions are added by the services as needed.

Chapter 7
Future Work

While the security system provides the functionality needed to implement higher level services and functionality. The service framework provides security authentication for the services provided that the keys for the system are properly protected. Currently the keys need to be protected using the methods of the operating system. These methods can be limited in their protection schemes, for instance if the keys are stored using the Network File System (NFS) the private keys would be vulnerable to generic network snooping. It might be possible to give passwords for the keys and/or require the keys to be stored on secure removable storage devices like iButtons or USB keys. How such a method of key storage would integrate with other non-standard ways of logging into a system, like voice authentication, is unknown.

Work also needs to be done on more novel methods of authentication like audio or visual recognition of users. Currently locally authentication of users is limited to user name/password combinations, iButton, or fingerprint scanners. While these methods are interesting, they do not start to provide the breadth of services needed in the full ACE environment.

It would also be useful to allow non-Java programs to provide or access services. Currently those services are limited to using a proxy for access. To make ACE accessible to all languages, the message system would need to be re-implemented using a method like XML-RPC [21] instead of serialized Java objects. Some experimentation with these methods occurred early in the project but they were not completed due to complexity issues in the software.

Finally the system as describes provides no way for one to store data in a secure way. A secure distributed file system need to be invented that would allow users a long term storage system. Such a system would need to store the data in multiple locations and encrypt the data when storing it. The storage encryption should be per user and not per server. Currently, no available file systems meets all the requirements for ACE.

Chapter 8
Accomplishments

This thesis describes the following accomplishments that were made for the creation of this section of the ACE project:

- The implemented Service Directory combines the features of the JINI lookup service and the Ninja Secure Directory Service in such a way to allow for services to be located by both their class (or sub-classes) and their current location in the environment.

- An authentication system that depends on the user's identity and not the identity of the machine he or she is running the application on. This is an improvement over the standard RMI authentication techniques where a user can only be identified by his host or by his port. It also allows for clients to authenticate servers in the same manner as clients

- The authorization system using the Keynote Trust Management System. The authentication system allows for more fine grained control of what actions each user is allowed to perform and to allow. It also allows RMI type services to have an integrated security model at the method level instead of at the object level.

- A set of services that allow for the authentication and authorization services to be used and updated over the network. The Authentication Database and the User Database let the services discover their permissions on the fly instead of being given their permissions at the start.

- A drop in replacement for the RMI architecture. The new Enhanced RMI service provides interactions and interfaces similar to the standard RMI model as well as letting standard RMI and Enhanced RMI services interact without difficulty.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis. “The Role of Trust Management in Distributed System Security.”, Secure Internet Programming: Security Issues for Mobile and Distributed Objects, Springer-Verlag, 1999.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis,A. Keromytis, “The KeyNote Trust-Management System Version 2”, RFC 2704, September 1999.

[3] Czerwinski, Zhao, Hodes, Joseph, Katz,“An Architecture for a Secure Service Discovery Service”, Fifth Annual International Conference on Mobile Computing and Networks (MobiCom ’99), Seattle, WA, August 1999, pp. 24-35.

[4] Dallas Semiconductor Corp. “iButton Home Page”, www.ibutton.com.

[5] T. Dierks, C. Allen, “The TLS Protocol Version 1.0”, RFC 2246, January 1999.

[6] D. Eastlake, P. Jones, “US Secure Hash Algorithm 1 (SHA1)”, RFC 3174, September 2001.

[7] R. Housley, W. Polk, W. Ford, D. Solo “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, RFC 3280, April 2002.

[8] James Gosling, Bill Joy, Guy Steelem, Gilad Bracha “The Java Language Specification Second Edition”, 2000, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[9] B. Kaliski, “PKCS #1: RSA Encryption Version 1.5”, RFC 2313, March 1998.

[10] B. Kaliski, “PKCS #5: Password-Based Cryptography Specification Version 2.0”, RFC 2898, September 2000.

[11] G. Minden, J. Evans, “Ambient Computational Environments Research Description.’

[12] National Institute of Standards and Technology (NIST), “Specification for the Advanced Encryption Standard (AES)”, FIPS-197, November 26 2001.

[13] National Institute of Standards and Technology (NIST), “DES Modem of Operation”, FIPS 81, December 1980.

[14] H. Schulzrinne, GMD Fokus, S. Casner, R. Frederick, V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications”, RFC 1889, January 1996.

[15] Sun Microsystems, “Java Media Framework API”, http://java.sun.com/products/java-media/jmf/index.jsp.

[16] Sun Microsystems, “Jini Network Technology”, http://java.sun.com/developer/products/jini/index.jsp.

[17] Sun Microsystems, “Java Remote Method Invocation (Java RMI)”, http://java.sun.com/products/jdk/rmi/.

[18] Sun Microsystems, Inc., “Java Remote Method Invocation - Distributed Computing for Java”, http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html.

[19] UC Berkley, “The Ninja Project”, http://ninja.cs.berkeley.edu

[20] Matt Welsh ,“NinjaRMI: A Free Java RMI”,http://www.eecs.harvard.edu/ mdw/proj/old/ninja/ninjarmi.html.

[21] Winer, D., ”XML-RPC Specification”, January 1999, http://www.xmlrpc.com/spec

Appendix A
Available Services

The following services have been implemented using the ACE framework. These services are available in the 1.0 release of the ace3. Each of these services is either a level or virtual level of the Service Hierarchy.

A.1 Basic Service Levels

The following implement no services per se, but they provide basic structure in which other services can be extended.

A.1.1 Base

The Base level exists in order to allow for future expansion. The Base level extends directly off of the **SecureUnicastRemoteServer**. The Base level provides no methods or unique services.

A.1.2 Service

The Service level provides the methods needed for services to communicate and operate with clients The level provides methods that allow the services to reset, shutdown, security commands. The Service level extends directly off of the Base level.

A.1.3 Media

The Media level provides the base for all the services that implement audio and video transmission and receiving. This level provides methods for connecting the session, sharing keys, setting up session, and starting or stopping sessions. The level extends directly off of the Service level.

A.1.4 Device

The Device level provides all the methods needed for controlling devices. The level consists of methods like device turn on, device turn off, and device reset. The Device level extends off of the Service level. The implementation of the level is declared abstract, since there is no generic way to implement the power control of the device.

A.1.5 Database

The Database level extends directly off of the Service level. It is the connecting level for all the data storage services. The level only provides one method, flush, that empties out the database.

A.1.6 SQL Database

SQL Database is a virtual level that extends off of the Database. This level implements the functions required to access a SQL database. This include connection pools, SQL string encoding, SQL command execution. It provides no exposed methods, but only exists to help simplify the implementation of other services which want to use SQL databases.

A.1.7 ID Monitor

The ID Monitor extends off of the Device level. It exists off of the Device level to provide the implementation for users to login to the ACE. The ID monitor is extended by devices like a Fingerprint Identification Unit or a iButton authentication service. It provides methods to store user ID's and the commands they want executed when they login to the ACE.

A.2 Core Services

The core services exist to provide the basic information for all the other services to work properly. These services allow for services to find each other, find information about the environment, and to discover authentication within the environment.

A.2.1 Service Directory

The Service Directory extends of the SQL Database level. The service stores information about the services within the environment. See Section 3.1.3.1 for more information.

A.2.2 User Database

The User Database store information about the users in the environment and what information can be used to identify them. The level extends off of the SQL Database level. See Section 3.1.3.2 for more information.

A.2.3 Room Database

The Room Database contains information about rooms and what services are located inside of the rooms. The services listed are those bound to the room by physical location. See Section 3.1.3.3 for more information. The Room Database extends off of the SQL Database level.

A.2.4 Authentication Database

The Authentication Database holds the assertions for the environment. The service extends off of the SQL Database level. For more information, see Section 3.1.3.4

A.3 Devices

The following devices represent hardware devices that can be accessed within the environment.

A.3.1 Projector

The Projector level extends off of the Device level. It contains methods for accessing projector functions like changing the input source and changing the zoom on the projector.

A.3.2 Epson Projector

The Epson Projector implements a service for controlling an Epson 7350 projector. The level extends off of the Projector level and implements all the functionality of the Projector level plus control for the picture-in-picture functions available on the Epson 7350. The service communicates with the device by way of a serial connection that is accessed by using native interfaces.

A.3.3 PTZ Camera

The Pan-Tilt-Zoom (PTZ) Camera service provides all the methods needed to control a PTZ camera. The PTZ camera service provides movement and zoom methods with coordinates related to the camera position and positions located inside of the room. The PTZ level extends the Device level.

A.3.4 VCC3 Camera

The VCC3 Camera extends off of the PTZ Camera level and implements a service for controlling the Canon VCC3 PTZ camera and the Canon VCC4 camera running with VCC3 compatibility. The level is a virtual level, since the VCC3 camera provides no capabilities that are not exposed by the generic PTZ camera level. The service communicates with the camera is over a serial interface, implemented using native, non-Java code.

A.3.5 iButton Monitor

The iButton Monitor is an ID Monitor that listens to a 1-Wire interface[4] that can identify unique tokens. The service uses the Java based One Wire API provided by the iButton's maker, Dallas Semiconductor.

A.4 Media Services

The following service implement the Audio and Video transmission services within the ACE. They all utilize the Media Channels to move video from one service to another, instead of the usual Control Channels.

A.4.1 Audio Receive

The Audio Receive service takes in audio from another source and plays it out of the audio cards in the system. The service uses the Java Media Framework [15] to process and play the sounds. The service extends off of the Media level.

A.4.2 Audio Transmit

The Audio Transmit service captures audio from a microphone and transmit the audio to another service for processing or playing. The service uses the Java Media Framework [15] to capture the sound and process it into a form suitable for transmission. This level extends off of the Media level.

A.4.3 Audio/Video Receive

This service is for transmission of synchronous audio and video. It utilized the same Java Media Framework [15] functions as the Audio and Video services. The level extends off of the Media level.

A.4.4 Audio/Video Transmit

This service captures audio and video, processes them, and transmits both streams in such a away that the receiver can play them in sync. The service utilizes the same Java Media Framework [15] functions as the Audio and Video services. The level extends off of the Media level.

A.4.5 Converter

A Converter takes data from one Media service and converts the data into another form for processing by another Media service. The Converter uses the Java Media Framework [15] and extends off of the Media level.

A.4.6 Video Transmit

The Video Transmit service implements a service for capturing video from a device using the Java Media Framework [15]. The video is then processes and transmitted to a receiving service or converter. This service level extends off of the Media level.

A.4.7 Video Receive

The Video Receive service implements a service for displaying video using the Java Media Framework [15]. The service extends off of the Media level of the framework.

A.5 Other Services

The following services have been implemented but do not fit into the model of the other services. These services would be useful to create higher level functions, but currently are not used.

A.5.1 Network Logger

The Network logging service takes strings from other services and logs them to a specified file. The Network Logger extends the Service Level.

A.5.2 Host App Launcher

The Host App launcher takes commands from a specified user and runs a command as that user. The Host App Launcher runs the command on the local machine as the user who requested. The Host App Launcher extends off of the Service Level.

A.5.3 Host Resource Monitor

The Host Resource Monitor extends off of the Service level. It looks at the current machine and reports the current status of the processor load, memory usage, etc. This information can be used to help distribute services within the local environment.

A.5.4 System Resource Monitor

The System resource monitor extends off of the Service level. It takes information from the Host Resource Monitors and correlates the information to allow one to pick the best host and to get system level views of the whole status of the system.

Appendix B
Implementing a Service

The implementation of a service occurs in two phases. The first part consists of defining the interface, while the second part concerns the actual implementation of the service. After that point any clients that want to use the service can be implemented. The following chapter shows an example on how to implement a service. For an example, an implementation of a projector service that has a has a picture-in-picture (PNP) capability as well as the normal projector functions.

B.1 Interface Implementation

The first step to describe the interface into the service. This interface is implemented by both the server and the generated stubs. It allows the clients to access the stubs as if they were a local object running inside of the same JVM. All methods in an interface need to throw the **java.rmi.RemoteException** even if the method will not throw an exception normally. The **java.rmi.RemoteException** is used to both communicate normal exceptions in the server and to communicate communication errors. The implementations for ACE must also be members of the **edu.ku.ittc.ACE.Interface** package.

B.1.1 Projector Controls

The following is the interface for the projector class of services. It extends off the interface Device, and as such it borrows the definition for turning the projector on and off from that service. The interface defines the strings needed to identify the possible input sources and the methods to change the input sources of the projector.

```
package edu.ku.ittc.ACE.Interface;
import java.rmi.RemoteException;
```

```
public interface Projector
  extends Device {
     /** Possible projector input sources */
    public static final String PC1 = "PC1";
    public static final String PC2 = "PC2";
    public static final String RCA = "RCA";
    public static final String S_VIDEO = "S_VIDEO";
    public static final String BNC_RBG = "BNC_RGB";
    public static final String BNC_CRCYCB = "BNC_CRCYCB";
    public static final String COMPOSITE = RCA;

    /** Sets the input sources to one of the above */
    public void setVideoInputSource( String input )
       throws RemoteException;
    /** Gets the current input source */
    public String getVideoInputSource()
       throws RemoteException;
}
```

B.1.2 EpsonProjector Controls

The following is the definition for implementing the picture-in-picture (PNP) functionality that the Epson Projectors provide. The interface provides methods for changing the input sources of the PNP function. It also provides a new string that allows the user to change the functionality of the projector to off.

```
package edu.ku.ittc.ACE.Interface;
import java.rmi.RemoteException;

public interface EpsonProjector
  extends Projector {
    /** Video mode for turning PNP off. */
    public static final String PNP_OFF = "OFF";

    /** Sets input source and turns pnp on */
    public void setPictureInPicture( String Input )
      throws RemoteException;
   /** Gets status of pnp */
    public String getPictureInPicture()
      throws RemoteException;
}
```

B.2 Service Implementation

The following gives an example of the services that implement the Projector and EpsonProjector interface. These two services called, Projector and EpsonProjector are members of the **edu.ku.ittc.ACE.Service** package. Unlike the above interfaces, the services are not required to be members of the above class.

It is important to note that the constructors of both services throw the RemoteException. This is inherited from the constructor of the **edu.ku.ittc.ACE.Service.SecureUnicastRemoteObject** which is ultimately inherited by at root of the implementation tree. It is also important to note that the constants from the interfaces are not included int the implementation, since they are imported when the interface is used.

B.2.1 Projector Service

The following is the implementation of the Projector Service. While not directly necessary, since it only defines abstract methods, it is defined in order to keep the services matched with the interface tree. A different implementation could omit this implementation and just implement the Epson Projector service directly. The service extends off of the Device service an as such it inherits all the methods of the Base, Service, and Device interfaces and implementations of the superclass services, as well as the Projector interface that it directly inherits.

```
package edu.ku.ittc.ACE.Service;

import java.rmi.*;
import java.rmi.server.*;
import edu.ku.ittc.ACE.Library.*;

public abstract class Projector
  extends Device
  implements edu.ku.ittc.ACE.Interface.Projector {
    public Projector() throws RemoteException {

    }

    public abstract void setVideoInputSource( String input )
      throws RemoteException;
    public abstract String getVideoInputSource()
      throws RemoteException;
}
```

B.2.2 EpsonProjector Service

The following is the implementation of the Epson Projector service. It uses the **EpsonProjector7350** object as the means to communicate with the projector.

```
package edu.ku.ittc.ACE.Service;

import java.rmi.*;
import java.rmi.server.*;
import edu.ku.ittc.ACE.Library.*;
import java.util.*;
import edu.ku.ittc.ACE.EpsonProjector7350.*;

/**
   Implementation for service.
 */
public class EpsonProjector
    extends Projector
    implements edu.ku.ittc.ACE.Interface.EpsonProjector {
```

The following defines the service variables. The first one defines the the configuration key needed to retrieve the serial port to use. The second one defines the object reference to hold the projector object that the rest of the methods use to communicate with the projector.

```
  /** Key for the serial port config */
  private static final String PROJECTOR_SERIAL_PORT =
    "edu.ku.ittc.ACE.Service.EpsonProjector.SerialPort";

  /** Instance for projector */
  private ACEEpsonProjector MyProjector;
```

The following is the method called when the service is started. It needs to create a new service object and then exit. If more complex setup needs to be done before the service is started, then it should be done in this method. In this case, nothing else needs to be done.

```
  /** Program to start service */
  public static void main( String Args[] ) {
    try {
      EpsonProjector x = new EpsonProjector();
    }
    catch( RemoteException E ) {
      E.printStackTrace();
    }
  }
```

The following is the constructor for the service. The service starts with an implicit call to its parent constructor with no arguments. The parents and grandparents handle exporting the object and registering it with the rest of the environment. After the parent constructor returns, the service gets the serial port from the **edu.ku.ittc.ACE.Library.Configuration** library and the initializes the projector service. If the projector cannot be started it exits the JVM.

```
  public EpsonProjector() throws RemoteException {
    try {
      String Port =
Configuration.getConfigValue( PROJECTOR_SERIAL_PORT ).trim();
      debug( "Serial Port: " + Port );
      MyProjector = new ACEEpsonProjector( Port );
      debug( "Projector Opened" );
    }
    catch( ACEEpsonProjectorException E ) {
      E.printStackTrace();
      // Can't open serial port so just die.
      System.exit( -1 );
    }
  }
```

The following methods implement the abstract methods from the Device level. Since these are specific to the device begin controlled, they were left to the leaf node to implement. Note the reset method, since there is no way to “reset” an Epson 3705 projector, this method does nothing.

```
  public boolean getPowerState() throws RemoteException {
    try {
      int PowerState = MyProjector.getPower();
      if(PowerState == ACEProjectorInterface.POWER_ON) {
return true;
      }
      else {
return false;
      }
    }
    catch( ACEEpsonProjectorException e ) {
      e.printStackTrace();
      throw new RemoteException( e.getMessage, e );
    }
  }

  public void resetDevice()
    throws RemoteException {
    // Do nothing.
  }

  public void powerOn()
    throws RemoteException {
    try {
      MyProjector.setPowerOn();
    }
    catch( ACEEpsonProjectorException e )
      {
e.printStackTrace();
throw new RemoteException( e.getMessage, e );
      }
  }

  public void powerOff()
    throws RemoteException {
    try {
      MyProjector.setPowerOff();
    }
    catch( ACEEpsonProjectorException e ) {
      e.printStackTrace();
      throw new RemoteException( e.getMessage, e );
    }
  }
```

The following methods implement the methods inherited from the Projector interface.

```
  public void setVideoInputSource( String input )
    throws RemoteException {
    try {
      if(input.equals(PC1)) {
MyProjector.setInputSource(
   ACEProjectorInterface.INPUT_PC1 );
      }
      else if(input.equals(PC2)) {
MyProjector.setInputSource(
   ACEProjectorInterface.INPUT_PC2 );
      }
      else if(input.equals(RCA)) {
MyProjector.setInputSource(
   ACEProjectorInterface.INPUT_RCA );
      }
      else if(input.equals(S_VIDEO)) {
MyProjector.setInputSource(
   ACEProjectorInterface.INPUT_S_VIDEO );
      }
      else if(input.equals(BNC_RBG)) {
MyProjector.setInputSource(
   ACEProjectorInterface.INPUT_COMPONENT );
      }
      else if(input.equals(BNC_CRCYCB)) {
MyProjector.setInputSource(
   ACEProjectorInterface.INPUT_COMPOSITE );
      }
    }
    catch( ACEEpsonProjectorException e ) {
      e.printStackTrace();
      throw new RemoteException( e.getMessage, e );
    }
  }

  public String getVideoInputSource()
    throws RemoteException {
    try {
      int Source = MyProjector.getInputSource();
      String ArgValue = "";
      switch(Source)
{
case ACEProjectorInterface.INPUT_PC1 :
  ArgValue = PC1;
  break;
case ACEProjectorInterface.INPUT_PC2 :
  ArgValue = PC2;
  break;
```

```
case ACEProjectorInterface.INPUT_RCA :
  ArgValue = RCA;
  break;
case ACEProjectorInterface.INPUT_S_VIDEO :
  ArgValue = S_VIDEO;
  break;
case ACEProjectorInterface.INPUT_COMPOSITE :
  ArgValue = BNC_RBG;
  break;
case ACEProjectorInterface.INPUT_COMPONENT :
  ArgValue = BNC_CRCYCB;
  break;
default: ArgValue = null;
}
      return ArgValue;
    }
    catch( ACEEpsonProjectorException e ) {
      e.printStackTrace();
      throw new RemoteException( e.getMessage, e );
    }
  }
```

The final set of methods implement the methods gained from directly implementing the **EpsonProjector** interface.

```
  public void setPictureInPicture( String Input )
    throws RemoteException {
    try {
      if(Input.equals(RCA)) {
MyProjector.setPNPSource(
   ACEProjectorInterface.PNP_COMPOSITE );
      }
      else if(Input.equals(S_VIDEO)) {
MyProjector.setPNPSource(
   ACEProjectorInterface.PNP_S_VIDEO );
      }
      else if(Input.equals(PNP_OFF)) {
MyProjector.setPNPSource(
   ACEProjectorInterface.PNP_OFF );
      }
    }
    catch( ACEEpsonProjectorException e ) {
      e.printStackTrace();
      throw new RemoteException( e.getMessage, e );
    }
  }

  public String getPictureInPicture()
    throws RemoteException {
    try {
      int Source = MyProjector.getPNPSource();

      String PNPSource = "fail";

      switch(Source) {
case ACEProjectorInterface.PNP_COMPOSITE:
PNPSource = RCA;
break;
      case ACEProjectorInterface.PNP_S_VIDEO:
PNPSource = S_VIDEO;
break;
      case ACEProjectorInterface.PNP_OFF:
```

```
PNPSource = PNP_OFF;
break;
      default : break;
      }
      return PNPSource;
    }
    catch( ACEEpsonProjectorException e ) {
      e.printStackTrace();
      throw new RemoteException( e.getMessage, e );
    }
    return null;
  }
}
```

B.3 Stub Generation

The following is an example of the generated source file from the StubGenerator. The StubGenerator takes the interface and then generates an implementation of the methods that can be used to communicate with the remote service. All stubs extend from the **edu.ku.ittc.ACE.Service.Stub** class and implement all the remote methods that a class inherits. The class below is an example of the stub generated for the EpsonProjector class.

```
/*
Generated by edu.ku.ittc.ACE.StubGenerator
   DO NOT EDIT
*/
package edu.ku.ittc.ACE.Interface;

public class EpsonProjector_StubACE
  extends edu.ku.ittc.ACE.Server.Stub
```

```
implements edu.ku.ittc.ACE.Interface.EpsonProjector {
```

The following is the constructor called by the service when generating the first stub. The initialized stub is passed to the rmiregistry and then passed to the services initialized.

```
  public EpsonProjector_StubACE(
     edu.ku.ittc.ACE.Server.SocketFactory sf,
     String host,
     int port ) {
    super( sf, host, port );
  }
```

The following is the implementation of the **setPictureInPicture** method. The method takes the arguments places them in an array of objects, retrieves the **java.lang.reflect.Method** object referring to this method, and then passes them to the **callMethod** method of the parent class that will package the items up and send them on. The exceptions are fatal, because they indicate something is horribly wrong with the classpath.

```
  public void setPictureInPicture(java.lang.String arg0)
    throws java.rmi.RemoteException {
    try {
      java.lang.Object[] o = new java.lang.Object[1];
      java.lang.Class[] c = new java.lang.Class[1];
      o[0] =  arg0;
      c[0] =  Class.forName( "java.lang.String");
      java.lang.reflect.Method m =
this.getClass().getMethod( "setPictureInPicture",
   c );
      java.lang.Object ret = callMethod( m, o );
    }
    catch( java.lang.NoSuchMethodException e ) {
      e.printStackTrace();
      System.exit( -1 );
    }
    catch( java.lang.ClassNotFoundException e ) {
      e.printStackTrace();
      System.exit( -1 );
    }
  }
```

The following is the implementation of the **getPictureInPicture** method. It is similar to the **setPictureInPicture** except that it gets the return from the **callMethod** call and returns that as the return of the object. If a Java primitive is returned instead of an object, the **Stub** has methods to convert the objects to privative in the appropriate manner.

```
  public java.lang.String getPictureInPicture()
    throws java.rmi.RemoteException {
    try {
      java.lang.Object[] o = new java.lang.Object[0];
      java.lang.Class[] c = new java.lang.Class[0];
      java.lang.reflect.Method m =
this.getClass().getMethod( "getPictureInPicture", c );
      java.lang.Object ret = callMethod( m, o );
      return (java.lang.String) ret;
    }
    catch( java.lang.NoSuchMethodException e ) {
      e.printStackTrace();
      System.exit( -1 );
    }
    return null;
  }
```

The following are the other methods implemented by Stub generator. The bodies of the methods have been removed from this paper. Since the following methods do not differ substantially from the above methods their bodies have been removed for compactness.

```
  public void setVideoInputSource(java.lang.String arg0)
    throws java.rmi.RemoteException
{...}

  public java.lang.String getVideoInputSource()
    throws java.rmi.RemoteException
{...}

  public boolean getPowerState()
    throws java.rmi.RemoteException
{...}

  public void powerOn()
    throws java.rmi.RemoteException
{...}

  public void powerOff()
    throws java.rmi.RemoteException
{...}

  public void shutdown()
    throws java.rmi.RemoteException,
   edu.ku.ittc.ACE.RobustService.RobustServiceException
{...}
```

```
  public void reset()
    throws java.rmi.RemoteException,
   edu.ku.ittc.ACE.RobustService.RobustServiceException
{...}

  public void close()
    throws java.rmi.RemoteException
{...}

  public void connect(edu.ku.ittc.ACE.Interface.Service arg0)
    throws java.rmi.RemoteException,
   edu.ku.ittc.ACE.RobustService.RobustServiceException
{...}

  public void testStatus()
    throws java.rmi.RemoteException
{...}

  public edu.ku.ittc.ACE.Library.UserInformation[]
getUsersList()
    throws java.rmi.RemoteException,
   edu.ku.ittc.ACE.RobustService.RobustServiceException
{...}

  public void
addCommandWatchNotify(java.lang.String arg0,
boolean arg1)
throws java.rmi.RemoteException,
   edu.ku.ittc.ACE.RobustService.RobustServiceException
{...}
```

```
  public void
deleteCommandWatchNotify(java.lang.String arg0)
throws java.rmi.RemoteException,
   edu.ku.ittc.ACE.RobustService.RobustServiceException
{...}

}
```

B.4 Startup Scripts

The final part of the implementation is the scripts used to start the services. These scripts set up the variable and class path as needed. The work by calling the parent level scripts to set up the variable. The important item is the use of the the DAEMON variable to control what service is started by the last script called in the chain.

The following is the script for the Projector level.

```
#!/bin/sh

TOPDIR=‘dirname $0‘/..
TOPDIR=‘cd $TOPDIR; pwd‘

export CLASSPATH=$CLASSPATH

if( [ -z $DAEMON ] ) then
    export DAEMON=edu.ku.ittc.ACE.Service.Projector
fi

$TOPDIR/bin/Device.sh $@
```

The following is the script to start the EpsonProjector.

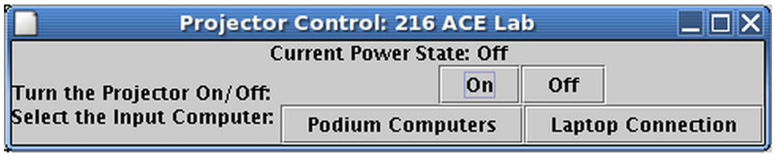

Projector Control: 216 ACE Lab
Current Power State: Off
Turn the Projector On/Off:
On
Off
Select the Input Computer:
Podium Computers
Laptop Connection

Figure B.1: A screen shot of a gui that utilizes the projector service

```
#!/bin/sh

TOPDIR=‘dirname $0‘/..
TOPDIR=‘cd $TOPDIR; pwd‘

export CLASSPATH=$CLASSPATH

if( [ -z $DAEMON ] ) then
    export DAEMON=edu.ku.ittc.ACE.Service.EpsonProjector
fi

$TOPDIR/bin/Projector.sh $@
```

B.5 Client Implementation

The following is the source code of a simple client that uses the projector service. The client connects to a projector service, as specified by the building name and room name passed to the service upon initialization. The service then contacts the Service Directory to get the room Information. Next the service contacts Room Database to get the correct room object for the service. Finally the client requires the room database to find the projectors that implement the required interface and are located in the specified room. The resulting service is then used to drive the GUI frame. Figure B.1 shows the resulting window shown the user.

```
package edu.ku.ittc.ACE.utility;
```

```
import edu.ku.ittc.ACE.Library.*;
import edu.ku.ittc.ACE.Interface.*;
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.rmi.*;

public class ProjectorFrame
  extends JPanel
  implements ActionListener {

  /** projector service */
  private Projector MyProjector = null;

  /** Strings used in the gui */
  private static final String ON_SYMBOL = "On";
  private static final String OFF_SYMBOL = "Off";
  private static final String WINDOWS_SYMBOL =
    "Podium Computers";
  private static final String LINUX_SYMBOL =
    "Laptop Connection";

  /** State variable */
  private boolean currentState = false;
  /** label for state */
  private JLabel PowerStateLabel;
```

The following is the main method of the object. This method connects to the projector service then creates an new window accessing the projector service. If it fails to do either, the method exits the JVM.

```
  public static void main( String Args[] ) {
    try {
      if( Args.length < 2 ) {
System.out.println( "Usage: ProjectorGUI " +
    "BuildingName RoomName" );
System.exit( -123 );
      }

      // get the projector
      Projector proj = null;
      proj = getProjector( Args[0], Args[1] );

      if( proj == null ) {
System.out.println( "Specified Projector " +
    "cannot be found." );
System.exit( -124 );
      }

      // Initialize the frame
      ProjectorFrame x = new ProjectorFrame( proj );

      // Finish the setup
      JFrame jf =
new JFrame( "Projector Control: " +  Args[1] );

      jf.getContentPane().add( x );
      // jf.setSize( 640, 480 );
      jf.pack();
      jf.setVisible( true );
      jf.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e)
    {System.exit(0);}
});
    }
    catch( RemoteException e ){
      e.printStackTrace();
      System.exit( -1 );
    }
  }
```

The following constructor sets up the window, creates the button and associates the listeners with the same frame. The buttons call the method, **actionPerformed**, when they are pressed. The method, **actionPerformed**, actually uses the Projector service, which is passed to the constructor as an argument.

```
  public ProjectorFrame( Projector p )
    throws RemoteException {

    this.MyProjector = p;
    BoxLayout x =
new BoxLayout( this, BoxLayout.Y_AXIS );
    this.setLayout( x );

    /* Power State panel */
    JPanel PowerStatus = new JPanel();
    BoxLayout y =
new BoxLayout( PowerStatus, BoxLayout.X_AXIS );
    PowerStatus.setLayout( y );
    PowerStatus.add( new JLabel( "Current Power State: " ) );
    PowerStateLabel = new JLabel();
    currentState = p.getPowerState();
    updatePowerStateLabel();
    PowerStatus.add( PowerStateLabel );

    /** Button Labels */
    JPanel VertItems1 = new JPanel();
    BoxLayout a =
new BoxLayout( VertItems1, BoxLayout.Y_AXIS );
    VertItems1.setLayout( a );
    VertItems1.add( new JLabel( "Turn the " +
   "Projector On/Off: " ) );
    VertItems1.add( new JLabel( "Select  " +
   "the Input Computer: " ) );

    // Buttons
    JPanel VertItems2 = new JPanel();
    BoxLayout a2 =
new BoxLayout( VertItems2, BoxLayout.Y_AXIS );
    VertItems2.setLayout( a2 );

    // power buttons
    JPanel PowerSelector = new JPanel();
    BoxLayout w =
new BoxLayout( PowerSelector, BoxLayout.X_AXIS );
    PowerSelector.setLayout( w );
    JButton PowerOn = new JButton( ON_SYMBOL );
    PowerOn.addActionListener( this );
    JButton PowerOff = new JButton( OFF_SYMBOL );
    PowerOff.addActionListener( this );
    PowerSelector.add( PowerOn );
    PowerSelector.add( PowerOff );

    // Input buttons
    JPanel InputSelector = new JPanel();
    BoxLayout z =
new BoxLayout( InputSelector, BoxLayout.X_AXIS );
    InputSelector.setLayout( z );
    JButton InputWindows = new JButton( WINDOWS_SYMBOL );
    InputWindows.addActionListener( this );
    JButton InputLinux = new JButton( LINUX_SYMBOL );
    InputLinux.addActionListener( this );
    InputSelector.add( InputWindows );
    InputSelector.add( InputLinux );

    VertItems2.add( PowerSelector );
    VertItems2.add( InputSelector );

    // Finished adding the buttons
    JPanel holder = new JPanel();
    BoxLayout bl = new BoxLayout( holder, BoxLayout.X_AXIS );
    holder.setLayout( bl );
    holder.add( VertItems1 );
    holder.add( VertItems2 );

    this.add( PowerStatus );
    this.add( holder );
  }
```

The following is the implementation of the method, **actionPerformed**. The code uses the projector service and calls methods on the service to turn the projector on or change the input source. The actual method used to communicate with the remote service is hidden here, the client only needs to handle errors thrown through the **java.rmi.RemoteException** exception.

```
  public void actionPerformed( ActionEvent a ) {
    try {
      System.out.println( a.getActionCommand() );
      Projector p = getProjector();
      // On action
      if( a.getActionCommand().equals( ON_SYMBOL ) ) {
debug( "Entering on" );
p.powerOn();
currentState = p.getPowerState();
updatePowerStateLabel();
debug( "Leaving on" );
      }
      // Off actions
      else if( a.getActionCommand().equals( OFF_SYMBOL ) ) {
debug( "Entering off" );
p.powerOff();
currentState = p.getPowerState();
updatePowerStateLabel();
debug( "Leaving off" );
      }
      // Windows Input
      else if
         ( a.getActionCommand().equals( WINDOWS_SYMBOL ) ) {
p.setVideoInputSource( Projector.PC1 );
      }

      // Linux input
      else if
         ( a.getActionCommand().equals( LINUX_SYMBOL ) ) {
p.setVideoInputSource( Projector.PC2 );
      }
    }
    catch( RemoteException e ) {
      e.printStackTrace();
    }
  }

  /** returns ref to projector */
  Projector getProjector() {
    return MyProjector;
  }

  /** debug statement */
  private static void debug( String d ) {
    System.out.println( d );
  }
```

The final method of note, takes the information about the Building and the room and gets a Projector service within the room. The client isn't too particular about which service it gets and if there is more than one service it will choose which ever one the service directory returns first. Before the service can contact the Projector service, it needs to communicate with the Room Database to get the object that represents the requested room. After that object is retrieved, the Service Directory can be contacted to retrieve the Projector service.

```
  /** gets the projector service */
  private static Projector getProjector( String BuildingName,
 String RoomName )
    throws RemoteException  {
```

```
    // Get the room database.
    RoomDatabase rd = null;

    // Need ref to service directory.
    ServiceDirectory sd =
      ServiceAccess.getServiceDirectory();

    // Set services that are of class RoomDatabase
    ServiceAttributes sa[] =
      sd.getServices(
         new
         ClassHierarchy( "edu.ku.ittc.ACE.Interface.RoomDatabase" ) );

    // Check services for services one can connect to.
    if( sa != null ) {
for( int i=0; i < sa.length; i++ ) {
  try {
    Remote r = ServiceAccess.getService( sa[i] );
    if( r != null ) {
      rd = (RoomDatabase) r;
      break;
    }
  }
  catch( Exception e ) {
    e.printStackTrace();
    rd = null;
    continue;
  }
}
if( rd == null ) {
  System.err.println( "No Room Directory found. "+
              "Please start one." );
  System.exit( -1 );
}
else {
  debug( "RD: " + rd );
}
    }

    // Since I have a room directory, now try to get
    // specified room object.
    Room rm = rd.getRoom( BuildingName, RoomName );
    if( rm != null ){
      debug( rm.toString()  );
    }
    else {
      debug( "Room NULL" );
      System.err.println( "No room with the name " + RoomName +
  " was found in the building " +
  BuildingName +
  ". Please add it to the room database" );
      System.exit( 0 );
      return null;
    }

    // Now have room object, get the projectors
    // that implement the object get the first
    // one in the room.
    Projector proj = null;
    ServiceAttributes[]sa2 =
      sd.getServices( rm,
 new ClassHierarchy( "edu.ku.ittc.ACE.Interface.Projector" ) );

    debug( "sa length:" + sa2.length );
    if( sa2 != null ) {
for( int i=0; i < sa2.length; i++ ) {
  debug( sa2[i].toString() );
  try {
    debug( "Getting the projector" );
    Remote r = ServiceAccess.getService( sa2[i] );
    debug( "Got r" );
    if( r != null ) {
      debug( "Converting r to string" );
      debug( r.toString() );
    }
    else {
      debug( "Remote connection is null" );
    }
    debug( "Before class cast" );
    proj = (EpsonProjector) r ;
    proj.testStatus();
    debug( "Got the projector" );
  }
  catch( Exception e ) {
    proj = null;
    e.printStackTrace();
    continue;
  }
}
```

```
    }
    else {
      debug( "No projectors in the Service Directory" );
    }

    if( proj == null ) {
      System.err.println( "Cannot connect to the " +
                          "projector.  An error " +
           "as orrcured." );
    }
    return proj;
  }

  /** Updates the power label */
  private void updatePowerStateLabel() {
    if( currentState == false ) {
PowerStateLabel.setText( "Off" );
    }
    else {
      PowerStateLabel.setText( "On" );
    }
  }
}
```

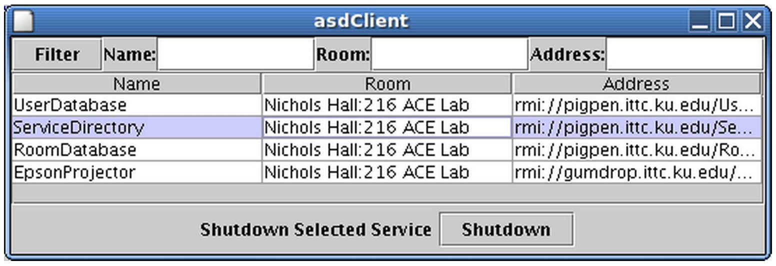
Appendix C
Using the Core Services

The following chapter describes the use of the Core services by other services and how one would use the services in their own clients. The core services are the Service Directory, the User Database, the Room Database, and the Authentication Database.

C.1 Service Directory

The Service Directory is the most important of the services and it is the only one contacted without the help of the Service Directory itself. The interface for Service Directory also comes along with the help of a class called **ServiceAccess**. A GUI has also been implemented in order to see the items currently stored in the Service Directory.

The interface into the Service Directory contains a number of methods to access the stored data. The methods are all called **getServices** and returns an array of objects of the **ServiceAttributes** class. The **ServiceAttributes** class contains the information to contact the service. The final set of methods are used by services to register services with the service directory. The registration and unregistrations methods are usually called by the services internally and not called by the services directly.



Name	Room	Address
UserDatabase	Nichols Hall:216 ACE Lab	rmi://pigpen.ittc.ku.edu/Us...
ServiceDirectory	Nichols Hall:216 ACE Lab	rmi://pigpen.ittc.ku.edu/Se...
RoomDatabase	Nichols Hall:216 ACE Lab	rmi://pigpen.ittc.ku.edu/Ro...
EpsonProjector	Nichols Hall:216 ACE Lab	rmi://gumdrop.ittc.ku.edu/...

Figure C.1: A screen shot of a gui that utilizes the Service Directory

A GUI called `asdClient.sh` has been implemented to give a view of the services currently stored in the Service Directory. The client allows for one to see a subset of the services and to select which service needs to be shutdown. Figure C.1 shows a screen shot of the service.

C.1.1 ServiceDirectory Interface

The following is the interface to use the Service Directory:

```
package edu.ku.ittc.ACE.Interface;


import java.util.*;
import edu.ku.ittc.ACE.Library.*;
import java.rmi.*;


public interface ServiceDirectory extends Database
{
```

The following methods search for services based on a number of different characteristics. All the methods return an array of **ServiceAttributes** classes.

```
    public ServiceAttributes[]
getServices( ClassHierarchy c )
throws RemoteException;

    public ServiceAttributes[]
getServices( String Name )
throws RemoteException;

    public ServiceAttributes[]
getServices( Machine m )
throws RemoteException;
```

```
    public ServiceAttributes[]
getServices( Room rm )
throws RemoteException ;

    public ServiceAttributes[]
getServices( String Name,
     ClassHierarchy c )
throws RemoteException;

    public ServiceAttributes[]
getServices( String Name,
             Machine m )
throws RemoteException;

    public ServiceAttributes[]
getServices( String Name,
             Room rm )
throws RemoteException;

    public ServiceAttributes[]
getServices( Machine m,
     ClassHierarchy c )
throws RemoteException;

    public ServiceAttributes[]
getServices( Room rm,
                     ClassHierarchy c )
throws RemoteException;
```

```
    public ServiceAttributes[]
        getServices( String Name,
             Room rm,
     ClassHierarchy c)
throws RemoteException;

    public ServiceAttributes[]
getServices( String Name,
      Machine m,
      ClassHierarchy c )
throws RemoteException;
```

The following methods are used internally by the services to register, unregister, and update their attributes.

```
    public void register( ServiceAttributes Attribs )
throws RemoteException;
    public void unregister( ServiceAttributes servAttrs )
throws RemoteException;
    public void renew( ServiceAttributes Attribs )
throws RemoteException;
    public long getLeaseTime()
throws RemoteException;
    public void
         addServiceDirectoryListener( ServiceDirectoryListener
                                              listener)
        throws RemoteException;
}
```

C.1.2 ServiceAccess Helper Class

The **ServiceAccess** helper class is used to get the initial connection to the Service Directory and to create new connections to the services based on their **ServiceAt-**

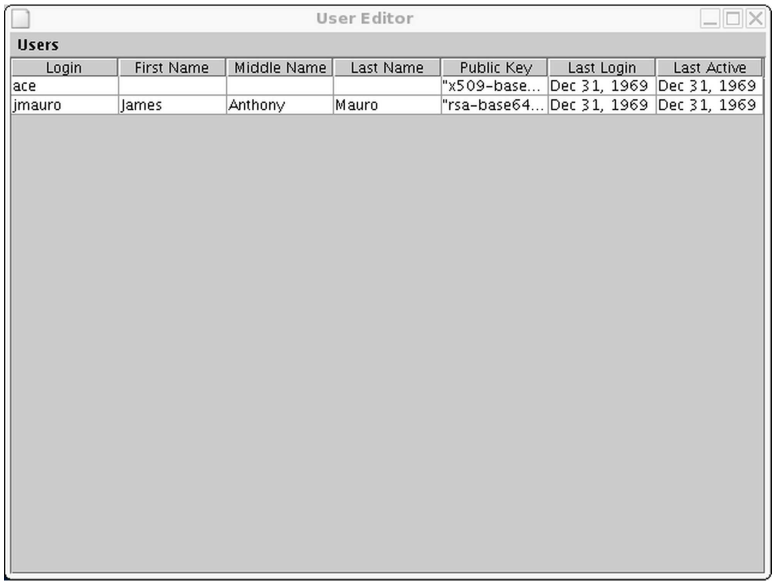
tributes.

```
package edu.ku.ittc.ACE.Interface;

/**
   Finds the instance of the ASD and returns
   that instance to the user
*/
public class ServiceAccess implements RobustServiceFinder {
  /** Gets the service directory */
  public static ServiceDirectory
      getServiceDirectory()
   {...}

  /**
    @return An instance of a service interface.
    If the service does not really
    exist then return a NULL.
   */
    public static Service
      getService ( ServiceAttributes servAttr )
    {...}
}
```

C.2 User Database

The User Database stores information about the users and the items needed to identify them. The database stores in the information in two objects, **UserAttribues** and **UserCharacteristics**. A GUI called `User Editor` has been created to allow the data in the database to be updated and changed.

The User Database's interface allows for a user to be entered by creating a **User-**



Login	First Name	Middle Name	Last Name	Public Key	Last Login	Last Active
ace				"x509-base...	Dec 31, 1969	Dec 31, 1969
jmauro	James	Anthony	Mauro	"rsa-base64...	Dec 31, 1969	Dec 31, 1969

Figure C.2: A screen shot of the current users stored in the User Database

Attributes class. This class contains the information about the user that corresponds to the user's public key, name, login name, and last login time. The database stores these objects until they are requested by the user. After a user has been entered his or her information can be updated by adding the new characteristics to the database. The characteristics are returned to the ID Monitors in terms of a **UserCharacteristics** class. This class contains the characteristics, the device the characteristics are intended for and the command to execute when the user is identified by the device. The command should be launched by the Host App Launcher and not by the ID Monitor its self.

The `User Editor` GUI allows for users to be entered into the system. The GUI consists of two main components. The initial screen (shown in figure C.2 shows the screen that allows for a user to be edited after he or she has been entered into the system. The second screen (shown in figure C.3) allows for an individual user to be entered into the system. For the second screen the user's public key is retrieved from the file system.

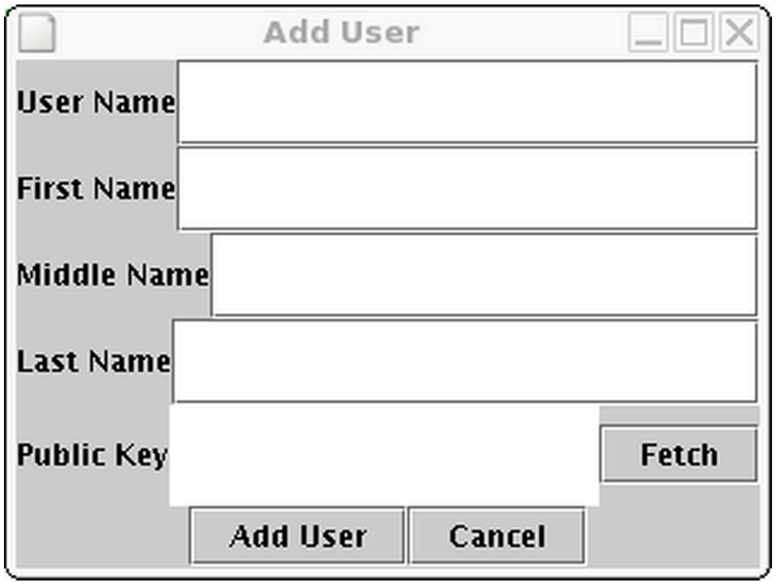


Figure C.3: A screen shot of the interface to add a new user.

C.2.1 User Attributes Class

The **UserAttributes** class contains the representation of the user within the ACE. It contains information about the users such as their public key, login name, real name and list time the user logged in.

```
package edu.ku.ittc.ACE.Library;

import java.util.*;

public class UserAttributes
implements java.io.Serializable {
```

The following constructor makes a new **UserAttributes class** containing the options as specified. Most of the objects are created by the `User Editor` or the User Database service and not by individual users.

```
  public UserAttributes(String key,
String user,
String firstName,
String lastName,
String middleName,
String userLoc,
Date logInTm,
Date activityTime)
  { ... }
```

The following methods allow for the stored objects to be retrieved.

```
  /** This is just the getter method for obtaining
      this user’s public key.
      @return The public key of the user.
   */
  public String getPublicKey()  { ... }

  /** This is just the getter method for obtaining
      this login name of this user.
      @return The login name of the user.
   */
  public String getLoginName()  { ... }

  /** This is just the getter method for obtaining
      the first name of this user.
      @return The first name of the user.
   */
  public String getFirstName()  { .. }

  /** This is just the getter method for obtaining
```

```
    the middle name of this user.
    @return The middle name of the user.
 */
public String getMiddleName()  { ... }

/** This is just the getter method for obtaining
    the last name of this user.
    @return The last name of the user.
 */
public String getLastName()  { ... }

/** This is just the getter method for obtaining
    the full name of this user.
    @return The full name of the user.
 */
public String getFullName()  { ... }

/** This is just the getter method for obtaining this user’s
    ID strings that are used to identify him/her on different
    ACE identification devices.
    @return The user’s ID strings used by the ACE devices.
 */
public Vector getUserIDs()  { ... }

/** This is just the getter method for
    obtaining this user’s ID device names that are
    used to identify for which ACE identification device a
    particular user ID string is for.
    @return The user’s ID device names.
 */
```

```
  public Vector getUserIDDevices()  { ... }

  /** This is just the getter method for returning
      the user’s current location.
      @return The user’s current location
      where he/she is active.
   */
  public String getUserLocation()  { ... }

  /** This is just the getter method for returning the
      time when the user logged into his/her current
      active location/host.
      @return The time when the user logged into
      his/her current location.
   */
  public Date getLogInTime()  { ... }

  /** This is just the getter method for returning
      the time when the user was last reported
      active at his/her current location.
      @return The time when the user was
      last reported active.
   */
  public Date getLastActiveTime()  { ... }
}
```

C.2.2 User Characteristics

The **UserCharacteristics** class allows for information that could be used to identify the user to a device to be transfered from the device into the User Database. The actual information stored is specific to the device and stored a string by the user database.

If a device needs to store a more complex, binary structure the device should encode the binary data into a string using an algorithm like base-64. The object also stores a command to be run by the App Host Launcher when the user is identified.

```
package edu.ku.ittc.ACE.Library;

import java.util.Vector;
import java.lang.reflect.*;

public class UserCharacteristics
  implements java.io.Serializable
{

  /** Allows instantiation of an empty object
      Also sets the name of the device that the characteristics
      are for */
  public UserCharacteristics( String characteristics,
      String key,
      String deviceName,
      String command )
  {
    userCharacteristics = characteristics;
    publicKey = key;
    this.deviceName = deviceName;
    this.command = command;
  }

  /** Returns the public key
      @return The public key object
   */
  public String getPublicKey()
  { ... }
  /** Returns the user characteristics
      @return the user characteristics object
   */
  public String getUserCharacteristics()
  { ... }

  /** Returns the name of the device
      @return The full name as a string
   */
  public String getDeviceName()
  { ... }

  public String getCommand()
  { ... }


}
```

C.2.3 User Database Interface

The following is the interface needed to access the User Database. The methods are divided into two sections. The first section concerns adding users and the second section concerns adding information about the user logging information.

```
package edu.ku.ittc.ACE.Interface;

import java.util.*;
import edu.ku.ittc.ACE.Library.*;
import java.rmi.*;

public interface UserDatabase extends Database
```

```
{
```

The following methods allows for adding and removing a user. Users are added by passing an initialized **UserAttributes** class to the system. The users are removed by specifying them as the public key.

```
  public void registerUser( UserAttributes userAttrs )
    throws RemoteException;

  public void unregisterUser( String PublicKey )
    throws RemoteException;
```

The following method can be used to search the database for the already entered users.

```
  /** This method shall create a command object
      and obtain information about users from the
      ACE User Database.  The parameters passed into
      this method are later used as wildcard information to
      the ACE User Database sql table queries.  The
      information returned here describes
      the users that match the input parameters.

     @param publicKey The public key of the user we are
                      looking for.
     @param loginName The login name or names we are
                      looking for.
     @param firstName The first name of the user(s) we are
                      looking for.
     @param middleName The middle name of the user(s) we are
                       looking for.
     @param lastName The last name of the user(s) we are
                     looking for.
     @param userID A user’s device identification
   information we are looking for.
     @param userIDDevice The ACE ID Device type that
                 corresponds to the userID string
                         we are looking for.
     @param userLocation The location where the user
                         is currently at.
     @param logInTime The time when the user logged into
                      where he/she is at.

     @return An vector of UserAttributesthat match the input
             information/criteria.

     @exception ServiceUnavailableException
   */
  public UserAttributes[] getUsers( String PublicKey,
    String LoginName,
    String FirstName,
    String MiddleName,
    String LastName,
    String UserID,
    String UserIDDevice,
    String UserLocation,
    String LoginTime )
    throws RemoteException;
```

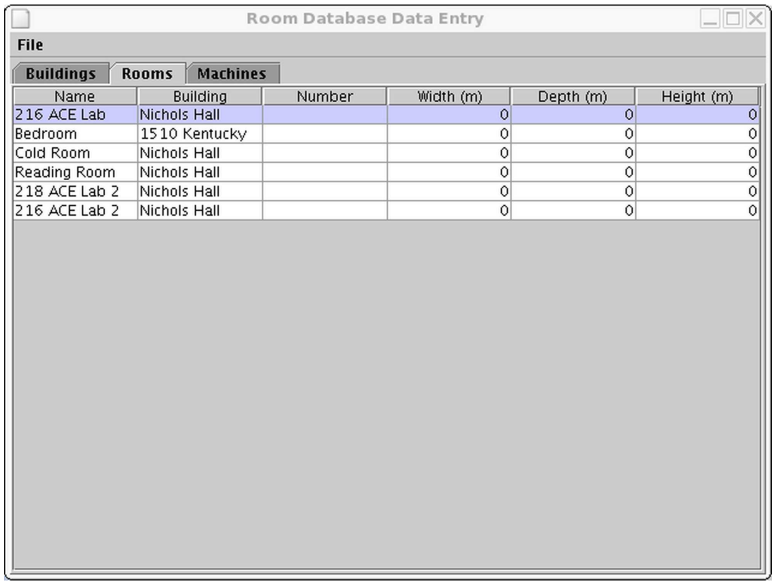
The following methods allow for a user’s record to be updated. The **updateUser** method updates a user inside of the database. The method matches the public key of the passed class to find the users stored record. The **updateUserLocation** allows the user’s last login location to be update.

```
  public void updateUser( UserAttributes attr )
```

```
    throws RemoteException;

  public void
updateUserLocation( UserAttributes attr,
                            String location,
    Date Login,
            Date Active )
    throws RemoteException;
```

The following methods are used by the ID Devices to store, delete and retrieve the **UserCharacteristics** class from the User Database.

```
FIXME : Get correct methods for retrieval.
  public void
    addUserIdentification( UserAttributes attr,
                  String UserID,
           String userIDDevice )
    throws RemoteException;

  public void
    removeUserIdentification(UserAttributes userAttribs,
                     String userID,
             String userIDDevice)
    throws RemoteException;
}

FIXME: Get the retrieve method
```



Name	Building	Number	Width (m)	Depth (m)	Height (m)
216 ACE Lab	Nichols Hall		0	0	0
Bedroom	1510 Kentucky		0	0	0
Cold Room	Nichols Hall		0	0	0
Reading Room	Nichols Hall		0	0	0
218 ACE Lab 2	Nichols Hall		0	0	0
216 ACE Lab 2	Nichols Hall		0	0	0

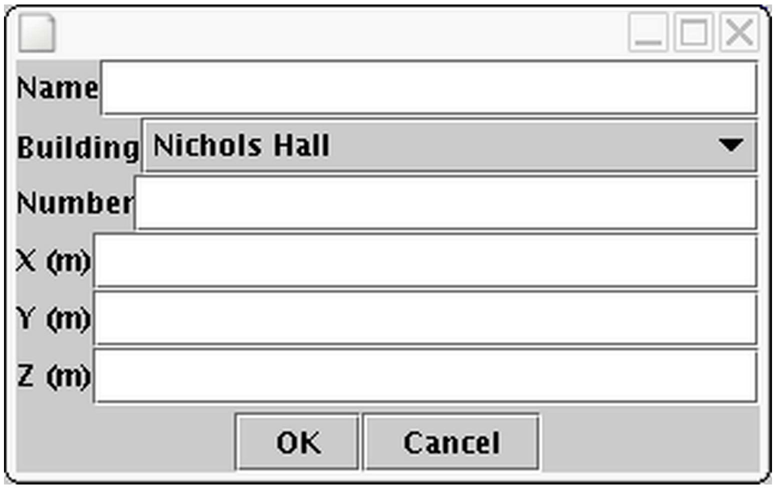
Figure C.4: A screen shot of the list of rooms stored in the database

C.3 Room Database

The Room Database is used to store information about the buildings, rooms, and machines stored in the room database. The information can be viewed and edited using the `rdbClient.sh`. The objects are viewed inside of the software as **edu.ku.ittc.ACE.Library.Building**, **edu.ku.ittc.ACE.Library.Room**, and **edu.ku.ittc.ACE.Library.Machine**. These objects are accessed by the room database.

The `rdbClient.sh` provides a way for a user to input the buildings, rooms, and machines currently stored in the Room Database. The main GUI consist of a tabbed set of windows that allows the user to view the stored objects. Figure C.4 is a screen shot of the GUI. The GUI provides an interface to enter new object (an example is shown in figure C.5).

A machine is added by using the `addMachine.sh` script. The script examines the machine and sets the items in the database appropriately.



Nichols Hall

Figure C.5: A screen shot of the interface to add a new room

C.3.1 Building Interface

The following is the interface for a building. The building is assumed to have an address as well as a name.

```
public interface Building
  extends java.io.Serializable
{
    public String getName();
    public String getStreetAddress();
    public String getZipCode();
    public String getCity();
    public String getState();
}
```

C.3.2 Room Interface

The following is the interface into a room. A room is assumed to have a name, a room number, be located in a building, and have a size. The size of the room is stored in a Point3d object where each point is the largest distance from the origin of the room.

The room's origin is the nearest corner to the door and all distances are in meters.

```
package edu.ku.ittc.ACE.Library;

import java.io.*;

/**
   Defines the interface for use of the ACE Room object.  This interface
   is used to compile classes that depend on the Room object before
   the room object actually compiled.
*/
public interface Room extends Serializable
{
  public Building getBuilding();
  public String getName();
  public String getNumber();
  public Point3d getSize();
  public String getRoomName();
  public String getBuildingName();
  public String toString();
}
```

C.3.3 Machine Interface

The following defines a machine in the ACE. A machine has a name and characteristics that go along with it including hardware, operating system, etc. It is also nominally located inside of a room. A machine is uniquely identified in the environment by its MAC address.

```
package edu.ku.ittc.ACE.Library;

public interface Machine
  extends java.io.Serializable {
```

```
  public String getName();
  public ServiceAttributes[] getServices();
  public Room getRoom();
  public java.net.InetAddress getInetAddress();
  public String getOperatingSystemType();
  public String getOperatingSystemVersion();
  public String getJavaVersion();
  public int getProcessorNumber();
  public int getProcessorSpeed();
  public int getBogoMIPS();
  public int getMemorySize();
  public String getVideoCardType();
  public int getNumberOfSerialPorts();
  public boolean canCaptureVideo();
  public boolean canCaptureAudio();
  public boolean canPlayAudio();
  public String getMACAddress();
}
```

C.3.4 RoomDatabase Interface

The following is the last interface into the Room Database. It defines the methods that the room database implements. These methods consist of searching and adding Machines, Rooms, and Buildings.

```
package edu.ku.ittc.ACE.Interface;

import edu.ku.ittc.ACE.Library.*;
import java.rmi.*;

public interface RoomDatabase extends Database {
```

The following methods define getting buildings from the database.

```
  public Building getBuilding( String BuildingName )
    throws RemoteException;
  public Building[] getBuildings()
    throws RemoteException;
```

The following methods add fetch rooms from the database.

```
  public Room getRoom( Building BuildingName, String RoomName )
    throws RemoteException;
  public Room getRoom( String BuildingName, String RoomName )
    throws RemoteException;
  public Room[] getRooms( Building BuildingName )
    throws RemoteException;
  public Room[] getRooms()
    throws RemoteException;
```

The following methods fetch machines from the database.

```
  public Machine getMachine( String Name, Room rm )
    throws RemoteException;
  public Machine getMachine( String Name )
    throws RemoteException;
  public Machine[] getMachines( Building BuildingID )
    throws RemoteException;
  public Machine[] getMachines( Room RoomID )
    throws RemoteException;
  public Machine[] getMachines()
    throws RemoteException;
```

These methods are used to add objects to the database.

```
  public void addBuilding( Building b )
    throws RemoteException;
  public void addRoom( Room r )
```

```
    throws RemoteException;
  public void addMachine( Machine m )
    throws RemoteException;
```

The following remove items from the database. A room and building cannot be removed if the machine or room exists.

```
  public void deleteMachine( Machine m )
    throws RemoteException;
  public void deleteRoom( Room r )
    throws RemoteException;
  public void deleteBuilding( Building b )
    throws RemoteException;
}
```

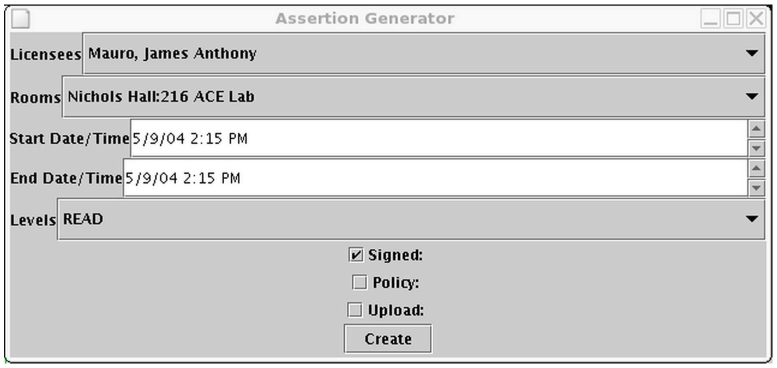
C.4 Authentication Database

The last core service is the Authentication Database. The Authentication Database stores assertions for the services. The assertions can be searched by service within the network and used to determine permissions within the network. These services. A GUI called `Cert Generator` was implemented to allow users to enter new assertions into the database. The interface for the database contains an interface for the keynote assertion and for the database itself.

The GUI uses the User and Room Database for determining the users and rooms that can be added to the system. The program then creates a new certificate. The new certificate is either saved to disk or entered into the database. Figure C.6 shows an screen shot of the GUI.

C.4.1 Keynote Assertion

A keynote assertion is an object representing a Keynote assertion. The assertion can be passed to the Authentication Database or a Keynote session.



Licensees Mauro, James Anthony

Rooms Nichols Hall:216 ACE Lab

Start Date/Time 5/9/04 2:15 PM

End Date/Time 5/9/04 2:15 PM

Levels READ

[✓] Signed:

[ ] Policy:

[ ] Upload:

Figure C.6: A screen shot of a gui that can be used to build new keynote assertions

```
package edu.ku.ittc.ACE.Keynote;

import java.util.*;

public class KeynoteAssertion
  implements java.io.Serializable
{
  private static final String APP_DOMAIN = "ACE";
  private static final String KEYNOTE_VERSION = "2";

  public KeynoteAssertion( String StringAssertion ) { ... }

  public KeynoteAssertion( String AssertionData,
   boolean trusted )
    throws KeynoteSignatureException,
   KeynoteSyntaxException { ... }



  public void sign( KeynoteKeyPair keys )  { ... }
  public boolean isSignatureVerified()  { ... }
}
```

C.4.2 Authentication Database Interface

The Authentication Database stores assertions until they are retrieved by a user. The assertions must be initialized keynote assertions.

```
package edu.ku.ittc.ACE.Interface;

import java.util.*;
import edu.ku.ittc.ACE.Library.*;
import java.rmi.*;
import edu.ku.ittc.ACE.Keynote.*;

public interface AuthenticationDatabase
  extends Database {
```

The following method adds an assertion to the database. An assertion is added with its characteristics to allow for a service to search for the assertion.

```
/** Adds an assertion to the database.  Assertion
    is connected to the the time between start
    and end, takes place in the specified room,
    and has the specified authorizers and licensees. */
  public void addAssertion(KeynoteAssertion assertion,
   Calendar startCalendar,
   Calendar endCalendar,
   Room rm,
   UserAttributes[] authorizers,
   UserAttributes[] Licensees )
    throws java.rmi.RemoteException ;
```

The following method retrieves the assertions as needed.

```
  /** Search for all the assertions that meet the
      requested statisics. These users are either listed
      in the licensee or authorizer fields.
  */
  public KeynoteAssertion[]
getAssertions(Calendar searchCalendar,
           Room rm,
      UserAttributes uas )
    throws java.rmi.RemoteException;

}
```

Appendix D
ACE Software and Setup

The following discusses the setup of the ACE tree, building the software, and setting up the initial system. This software is primarily written in the Java programming language, although some parts are written in C and accessed through the Java Native Interface System. The software was written to target the RedHat Linux 7.3 and RedHat Linux 9 systems with an installed Java Development Kit of 1.4 or later.

D.1 Source Tree Review

The source code is divided into two major sections. The first section, called third-party contains sources from other projects that are either not normally installed on the system or need to be modified for use within ACE. The src directory contains all the sources written for the ACE project. The src directory is further divided into drivers, java, and program directories. The drivers directory contains custom self-contained drivers for devices (like the Java interface for the Canon VCC3 camera.) The java directory contains the actual sources for what is considered to be ACE. The programs directory contains stand alone programs for accessing services and programs inside of ACE.

D.1.1 Third-Party Directory

The third party directory contains sources used by ACE, but not written by those working on ACE. They usually consist of drivers from the vendors and media libraries for MPEG4. The software is built along with the rest of ACE on a normal install, except for the rxtx and nasm directories, which must be installed separately before the build can take place.

The keynote directory is the software to run the Keynote Trust-Management Sys-

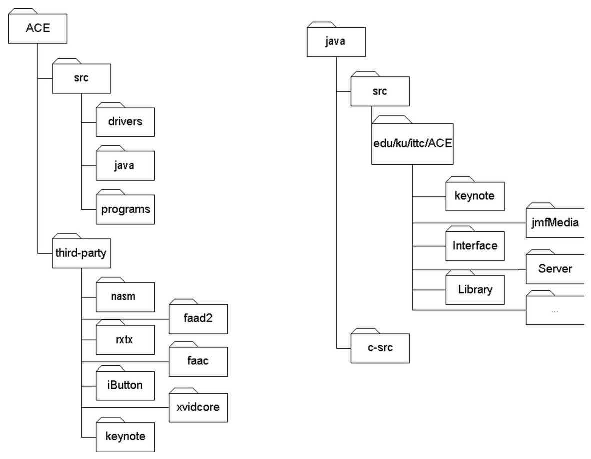


Figure D.1: Directory tree for the ACE software. The left tree is the overall tree from the root, while the right tree just shows the branches under the ACE/src/java directory where most of the project is housed.

tem. The iButton contains the software to access the iButtons under java. The mysql directory contains the Java database drivers for the MySQL Database. The jmf, libsndfile, xvidcore, ffmpeg, faad2, and faac directories contain the libraries for processing video and sound inside of java as well as the libraries for processing AAC audio and MPEG4 video.

The nasm and rxtx directories must be built separately of the tree. The nasm contains an i386 assembler used by some of the software libraries. The nasm version that ships with RedHat 7.3 and 9 does not support all the options required by the media libraries. The rxtx is the implementation of the Java Comm Serial Port library for Linux. Rxtx is required for the iButton libraries.

D.1.2 Src Directory

The src directory contains the software written to implement ACE. It is divided into three parts. The drivers sub-directory contains stand alone software to run certain devices. The Java sub-directory contains the implementation of ACE and the programs sub-directory contains programs implemented to access ACE and perform functions within ACE.

D.1.3 Drivers Sub-Directory

The drivers subdirectory contains drivers for the Epson 7350 Projector, the Sony Fingerprint Identification Unit (FIU), and the VCC3 camera. All the devices work over a serial interface and are implemented in the C library. A library called Serial is included as well. This software is used by the other libraries to implement the serial communication interface in a thread safe way.

D.1.4 Java Sub-Directory

The java sub-directory contains the code that makes up the core of ACE. This includes the service interfaces, service implementations, and any needed helper objects. The data is divided into two parts, the first part stored in the src directory contains the sources for the java portions of the project. The second directory called c-src contains the implementations for the native parts of the service.

D.1.4.1 src

The src directory contains the Java source code for ACE. The code all belongs to a sub-package under edu.ku.ittc.ACE structure. The sub-packages include drivers for Keynote, Epson 7350 Projector, Sony FIU, the Canon VCC3 camera and the Java Media Foundations. It also includes the service interfaces in the Interface directory and the implementation in the Service directory. The RobustService, StubGenerator, and Server directories implement the features which make ACE different from other RMI systems. Finally the Library and Room directories contain the generic objects that clients and services need to communicate complex information amongst each other.

All the code is stored in the directory related to the package name of the service and the file name of the implementation is related to the class name For instance the edu.ku.ittc.ACE.Service.ServiceDirectory class is implemented in a file called ServiceDirectory.java and stored in the edu/ku/ittc/ACE/Service directory under src.

D.1.4.2 c-src

The c-src directory contains the implementations of necessary native functions for src library. The native implementations are written in either C or C++. Implementations headers derive directly from the Java class files created by the `javah` program. The directory contains implementations for a number of JMF programs, Keynote, Sony FIU, VCC3 camera and the Epson 7350 projector.

D.1.5 Programs Sub-Directory

The final directory of note is the programs sub-directory. This directory contains the sources for programs that while not providing services directly to ACE, utilize ACE services to perform functions within the environment. These programs include a client to view the current Service Directory entries, a program to edit the User Database, a program to generate and store Keynote Assertions, and a GUI for the media programs.

D.2 How to Build the Source Tree

The source tree is designed to be built with one make command from the root. The build should be successful, if the correct libraries are already installed. The source tree builds its self and installs its files in the bin, etc, lib, and include. A script to build an installable RPM is included as well.

The system needs the following programs installed to build the tree:

- Java Development Kit version 1.4 or later (http://java.sun.com)
- GCC compiler version 2.95 or later (http://gcc.gnu.org)
- NASM version 0.98.36 or later (http://nasm.sourceforge.net)
- rxtx version 2.1.7 or later. (http://www.rxtx.org)The version that does not use the Java Comm libraries is required.
- Apache Ant version 1.6 or later (http://ant.apache.org)

- GNU Make version 3.79 or later (http://www.gnu.org)
- MySQL version 4 or later (http://www.mysql.com) Not needed for building, but required to run some services.

These programs need to be installed separately from building the ACE tree. The rxtx program must be installed into the JVM environment being used. To point the Makefiles to use the appropriate programs, one needs to edit the Makefile.global file at the root.

```
# JAVA Home Directory Directory
JAVA_HOME = /tools/java/i686/j2sdk1.4.2

## Ant control files
ANT_HOME=JAVA_HOME=$(JAVA_HOME) /users/jmauro/ant
ANT\_EXEC=$(ANT_HOME)/bin/ant -emacs
```

The three most important lines are above. The system uses make to control the main building and the compiling of all the C and C++ source files. Unfortunately make is not designed to compile Java programs. For building Java files, Apache Ant is used. It is called by the make files to build the Java programs. As such, make needs to be told both the location of the JDK which contains `javac` and the location and and program needed to run Ant.

After the environment has been setup, ACE can be built by running the following command at the root of the tree:

```
make install
```

ACE will then be installed into the correct directories on the root. If the program needs to be installed into another directory, the bin, etc, share, include, and lib directories can be directly copied.

There exists a spec file for building the ACE system on RPM based systems. A script called `makeACERPM.sh` (installed into the bin directory) can take the spec file and build a new RPM from the source tree. This RPM specifies that the software is installed into the /usr/local/ace directory.

D.3 Initial Setup

This section assumes that one installed ACE into the /usr/local/ACE directory. If that is not the case, please change the directories as needed.

D.3.1 System Setup

The following lists the commands and steps needed to setup ACE after it has been installed. It includes creating the “ace” user, configuring MySQL, setting the core services to start, setting up the Certificate Authority (CA) for all of ACE, keying the ACE user, then generating the main policy assertion for all of ACE.

D.3.1.1 Creating user “ace”

The first step is to create a user called, “ace”. This user is the “root” user of ACE. It should be created on all the machines in the environment and have the same home directory. The home directory is used to house specific files for ACE, like the master key, the Certificate Authority files and passwords for the services that use the MySQL database.

D.3.1.2 MySQL Setup

The core services in ACE all utilize the MySQL database for information storage and retrieval. The MySQL database needs to have databases created for each service and unique passwords for each service. The database only needs to be running on those hosts which are running a core service. A script called `configureMySQL.sh` has been created to help with the tasks.

The script takes one argument, which is the database being created (limited to either asd, roomdb, authdb or userdb). The script checks to see if the MySQL server is running. If the server is running the script creates a new database, a new user with a password stored in the ACE directory. For instance if one was creating the asd database, a database called ace_asd would be created, a user called ace_asd would be created to access the database and the password for the user is stored in ace/.mysqlPassword_ace_asd. The password should be unencrypted and be only readable to the ace user. If the password file does not exist and the `mkpasswd` program exists, a new password is generated. If `mkpasswd` does not exist, then the password file needs to be created by hand.

D.3.1.3 Service Setup

There are two main configuration files for ACE. The file, /usr/local/ace/etc/Daemon.conf, handles the configuration of the individual services. The second one The second one, /etc/ace/ace.conf, exists to control what services are started at boot. After the RPM is installed, these files need to be modified to indicate which services need to be started.

The following is an example of the /usr/local/ace/etc/Daemon.conf file:

```
asd=rmi://pigpen.ittc.ku.edu/ServiceDirectory
edu.ku.ittc.ACE.ACEDaemon.PolicyAssertions=$(TopDir)/etc/policyassert
edu.ku.ittc.ACE.Service.NetworkLogger.FileName=ACELogFile.log
edu.ku.ittc.ACE.Service.EpsonProjector.SerialPort=/dev/ttyS0
edu.ku.ittc.ACE.Service.VCC3Camera.SerialPort=/dev/ttyS1
edu.ku.ittc.ACE.Service.IButtonMonitor.DefaultPort=/dev/ttyS1
```

The example file shows the most commonly changed portions of the Daemon.conf. It include which serial ports the the devices are connected on, what is the location of the service directory and where the network logger logs to. The edu.ku.ittc.ACE.ACEDaemon.PolicyAssertions option tells the service where to look for its policy assertions. The file can take already stored variables and use them internally with the $\$VARIABLE$ name syntax. The variables can also be overridden on the command line using the -Dvariable=value syntax after the shell script to operate the program is executed. .

The following is an example of the /etc/ace/ace.conf:

```
ENABLE_ASD="yes"
ENABLE_ROOMDB="yes"
ENABLE_PROJECTOR="no"
ENABLE_USERDB="yes"
ENABLE_AUTHDB="yes"
ACE_INSTALL_PATH=/usr/local/ace
JAVA_HOME=/tools/java/i686/j2sdk1.4.2
ACE_USER=ace
```

This file is used by the `ACE` init script. The ENABLE_SERVICE lines tells the init script which service to start up on boot. “Yes” starts a services, while “no” or no answer causes the service not to start-up on boot. The ACE_INSTALL_PATH variable tell the script where to look for the startup and JAVA_HOME tells the script where the Java Virtual Machine is located.

D.3.1.4 Setting up the Certificate Authority

ACE needs a certificate authority to authorize the correct users within the environment. The CA resides in the ace user’s home directory. The CA can most simply be setup by using the `setupCA.sh` script which creates a new OpenSSL CA a directory called CA.

D.3.1.5 Keying the “ace” user

Please see the section D.3.2 for more information about creating ace’s key and adding him to the User Database. The actual keying of the user is no different than keying any other user, except that it needs to be done before the session starts.

D.3.1.6 Generating the Keynote Policy Assertion

The generic policy assertion for all of ACE needs to be created. The assertion can only be created after the keys for the "ace" user have been implemented. The basic policy assertion needs to be distributed on all the machines in the same ACE domain. It should be installed into the etc directory of ACE's install location. It should be given the name policyassertion. Since it is a policy assertion its authorizer must be policy and it does not need to be signed. The ACE user should be given _MAX_TRUST for the "ACE" application domain. Below is an example of a policy assertion:

```
keynote-version: 2
authorizer: POLICY
local-constants: KEY1 = "x509-base64:MIIEZzCC...LCSG0N2ICh"
licensees: KEY1
conditions: (APP_DOMAIN == "ACE") -> _MAX_TRUST;
```

D.3.2 User Setup

Setting up a user for working under the ACE environment occurs in three phases. The first phase has the user run the script, `setupAceUser.sh`, which generates the user's keys and request for the CA to sign his or her keys. The next setup has the administrator run `convertCerts.sh` with the user's name as an argument. The `convertCerts.sh` signs the user's certificate. Finally the `finishSetupAceUser.sh` script is run by the user. The `finishSetupAceUser.sh` finishes the setup by importing the signed certificate and the CA's certificate into the key rings for use under TLS and Keynote. The keys and their key rings are stored in the /.ace directory.