A Highly Adaptable Infrastructure for Service Discovery and Management in Ubiquitous Computing

Lalana Kagal, Vladimir Korolev, Sasikanth Avancha, Anupam Joshi, Timothy Finin, Yelena Yesha
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
email : {lkagal1,vkorol1,savanc1}@cs.umbc.edu
phone : 410-455-3971
fax : 410-455-3969

Abstract

In an age where wirelessly networked appliances and devices are becoming commonplace, there is a necessity for providing a standard interface to them that is easily accessible by any mobile user. The design outlined in this paper provides an infrastructure and communication protocol for presenting services to heterogeneous mobile clients in a physical space via some short range wireless links. This system uses a Communication Manager to communicate with the client devices. The Communication Manager can be modified easily to work with any type of communication medium, including TCP/IP, Infrared, CDPD and Bluetooth. All the components in our model use a language based on Extensible Markup Language (XML) giving it a uniform and easily adaptable interface. We explain our trade-offs in implementation and through experiments we show that the design is feasible and that it indeed provides a flexible structure for providing services. Centaurus defines a uniform infrastructure for heterogeneous services, both hardware and software , to be made available to diverse mobile users within a confined space.

1 Introduction

In the ubiquitous computing paradigm, information and services are accessible virtually anywhere and at any time via any device - phones, PDAs, laptops or even watches. The "SmartHome" and "SmartOffice" scenarios present a step towards realizing this vision. In particular, smart homes and offices consist of intelligent services that are accessible to users via handheld devices connected over short range wireless links. These "smart spaces" use sensors to gather information about the user and environment and help the user by suggesting relevant services. The intelligent services themselves will be more receptive to the user's requirements and use logical reasoning to provide better and more relevant support to individual users. The services will be integrated seamlessly into the environment that the user is familiar with, enabling easy and automatic usage. This is the vision that guides our research on the Centaurus system.

Our system is called Centaurus after the constellation which honors the Centaur Chiron, who was known as a wise teacher, healer and prophet. The goal is to design an infrastructure and communication protocol for providing services to a range of mobile clients in the smart spaces scenario. This framework is a part of our larger research program aimed at realizing ubiquitous computing systems that are composed of highly intelligent, articulate and social components. These components, known as dynamic negotiating agents, automatically become aware of each other and can exchange information to co-operatively provide services to the users. In particular, the idea of ad-hoc sets of entities that are dynamically formed to pursue individual and collective goals can be used to create the software infrastructure needed by the next generation of mobile applications. This infrastructure requires rethinking the neatly layered approach that separates networking, data management and user interface considerations, as our system design illustrates.

Centaurus consists of **Services, Clients, Communication Managers** and **Service Managers**. Communication Managers handle communication between various entities in the system. Service Managers are responsible for client and service registration, service leasing and service discovery. While within a confined space, the Client can access the services provided by the nearest Centaurus System (CS) via some short-range communication. The CS acts as an active proxy by executing services on behalf of any Client that requests them. This minimizes the resource consumption on the Client and also avoids having the services installed on each Client that wishes to use them, which is a blessing for most resource-poor mobile clients.

All Clients and Services need to communicate via Centaurus Communication Markup Language (CCML) (refer to section 5.3) which is based on Extensible Markup Language (XML). We found that this W3C Standard [7] is very useful in describing ontologies, and defining properties and interfaces of Services. As this is already being widely used, we think that it will help in integrating Centaurus with existing systems, including emerging semantic languages like DAML [26]. The Communication Manager is flexible and allows any medium to be used for communication, but for implementation purposes, we have used Infra-Red [12], CDPD [13] and Bluetooth [8]. Our framework is very robust and can recover easily from crashes due to the automatic state recovery.

This paper is organized as follows: Section 2 discusses other technologies, and compares Centaurus with them, Section 3 gives a brief overview of our design. The primary transport protocol used by the Centaurus system, the CentaurusComm Transport protocol, is discussed in Section 4. In Section 5 the design and modeling issues are covered. The actual implementation is detailed in Section 6, with the protocol illustrated in Section 7. Some experiments with the Centaurus system are described in Section 8. We present our conclusions in Section 9 and also discuss possible directions of our future research work.

2 Related Work

In the last couple of years, a number of technologies that deal with "SmartHomes" and "SmartOffices" have emerged. Among them are the Berkeley Ninja Project [1], the Portolano project[20] from the University of Washington, Stanford's Interactive Workspaces Project [22], Berkeley's Document-based Framework for Internet Application Control [4] and Active Spaces [5, 6] from University of Illinois at Urbana-Champaign.

The team at Stanford has developed hardware and software testbeds that include large display devices as well as personal mobile computing devices such as laptops and PDAs connected through a wireless LAN. They are creating an infrastructure for multiple users to communicate with multiple devices with the ability to move work between different devices.

University of Washington's Portolano project is in the early stages and is involved in "invisible computing" a term invented by Donald Norman [21] to describe ubiquitous computing, where devices supporting distributed services blend into the user's environment and become practically "invisible". The user would invoke these services not just by input but also through augmenting forms of interfacing like user movement, proximity of devices, identification tags, etc.

An Active space [5, 6] is a physical space including its different physical and virtual components, managed by an operating system, Gaia OS, which acts as a layer of abstraction over the particular properties of an Active Space. The Gaia OS manages the resources of an active space. Gaia does not define high-level policies regarding the behavior of the entities in the space. It concentrates on providing an infrastructure for the physical space and projecting a unified interface. This model, by insisting that the services be implemented as CORBA services, restricts the application developers. In Centaurus services in any language will be seamlessly integrated into the system, if they can use CCML (refer to section 5.3). Also, the Active Spaces project does not seem to be easily extended to support mobile users or different modes of communication i.e., Bluetooth, IR, CDPD and TCP. Centaurus has been built to allow flexibility in communication.

The Ninja project tries to link different services, through a range of devices ranging from PCs to cell phones and Personal Digital Assistants [1]. It has incorporated intelligence into the infrastructure and has the ability to adapt the content to a specific device.

Centaurus differs from Ninja in its service leasing abilities and state management. Unlike the Ninja project, Centaurus infrastructure delegates the state management to the Services themselves, with the Service Manager serving as the cache. The advantage of such approach is the decreased complexity of distributed state management and increased fault tolerance. Even in the event of Service Manager going down, it can recover easily because although it does not store any state information, the Services send it regular status to maintain their lease. Ninja uses a different security infrastructure. Currently we are working on the security aspect of the framework. We are using a system which combines Distributed Trust [24, 25] and Kerberos [16, 17]. It consists of a ticket granting server, that issues time-bound signed tickets for each mobile device. Ninja tends to concentrate on Web-based Services, whereas our system is able to support Services based on any platform, as long they can communicate with either the Service Manager through sockets, or one of the Communication Managers through the native

protocol and possess the ability to process Centaurus Capability Markup Language(CCML) messages (this is discussed in Section 3.4). We also do not distinguish between hardware and software Services, allowing the user to use either in the same way. Since all of the communication between Services and Clients in the Centaurus project are done with the use of XML, there is no need for complicated Operators and Paths used by the Ninja project to convert between different data representations.

Though both the Ninja project and Centaurus are aimed at providing a uniform infrastructure for a multitude of devices to use heterogeneous services, Centaurus is more applicable for 'SmartHomes' and 'SmartOffices' because of its independence of any kind of specific communication infrastructure allowing it to be easily implemented in the wide range of environments. In addition, Centaurus architecture is less prone to the failures of its components because of the use of multiple communication modules and automatic state recovery in the event of the Service Manager failure.

3 Overview

The main design goal of Centaurus is to develop a framework for building portals, using various types of mobile devices, to the world of "things" that users can communicate with and control. Centaurus provides a uniform infrastructure for heterogeneous services, both hardware and software services, to be made available to the users everywhere where they are needed.

Some of the key features of Centaurus are the following:

- The Service interface and communication protocols allow users to use different types of mobile devices for access. These devices may differ in the form of user interface, computation power, resource availability and size.
- The Centaurus infrastructure defines the Centaurus Capability Markup Language (CCML), based on XML, which provides a flexible and simple content description that enables the creation of content UI (not limited to Graphical User Interface only) which scales across a wide range of devices, like laptop, PDA, cellular phones etc. CCML also provides an effective means for describing service interfaces and client capabilities. Centaurus also defines the Centaurus Interface Definition Markup Language (CIML), which provides a machine-understandable description that allows applications to validate and interpret service descriptions and capabilities. This information can be used in various inference processes.
- The Service Manager is designed to be decentralized. Services can dynamically join and leave the system in a robust and flexible fashion.
- A requester of a Service can either be an end-user or another service which allows a service to be a composition of many services.
- Centaurus components use an efficient, message-object based transport protocol called the CentaurusComm Transport Protocol. In order to communicate using a specific media type, a communication module for that media type must be made part of the protocol stack. The CentaurusComm protocol allows multiple communication modules to coexist on the same node. The module for IR and a UDP-based module for Cellular Digital Packet Data (CDPD) have been completely developed. A preliminary, UDP-based module has been developed for Bluetooth.

4 CentaurusComm Transport Protocol

All components in the Centaurus system communicate using the CentaurusComm protocol. This section describes the protocol in detail.

Two primary modules, designated Level I and Level II protocol modules, form the core of the CentaurusComm protocol. The Level I module is highly dependent on the underlying communication medium whereas the Level II module is completely independent. However, both levels are dependent on the operating environment. At the time of writing of this paper, Level I modules for IrDA on PalmOS and Linux (kernel module), UDP on PalmOS and Linux (user space module) have been implemented. Level II modules are independent of the communication medium. These have been implemented for PalmOS and Linux. The version for Bluetooth on PalmOS is under development and a preliminary UDP-based Linux version is currently in use. A more improved version for Linux is under development.

4.1 System Architecture

CentaurusComm consists of the two protocol modules and an application program interface . The API is responsible for accepting the objects from the application layer for transmission and notifying it when messages are received.

The whole protocol is implemented as a collection of data structures and state machines. As the protocol is designed to run on a wide range of low power systems such as PDAs and low power embedded computers, it does not depend on any advanced operating system features such as signals and multithreading, that are typically not part of such systems. The protocol is designed in such a way that all the work that needs to be done by the protocol is done in small chunks, with each chunk lasting for a very short time. With the exception of domain name resolution that occurs very infrequently the protocol never blocks. Such a design gives an impression of concurrent execution of the user program and the protocol modules. The user program is responsible for calling a worker routine of the protocol.

The worker routine is part of the Level I module. Its main purpose is to perform message transmission or reception depending on transmit and receiver queues. Typical actions of this routine are: check “send” and “receive” queues, network connections, and the Level II state machine. If required the worker routine will attempt to send any data waiting in the “send” queue and/or run the Level II state machine to act upon data received from the “receive” queue. In addition, on peer-to-peer type networks, the worker routine will examine the table of outgoing messages and will trigger the Level II state machine in order to start the transmission of outgoing messages, if this event had not yet occurred. On master-slave type networks, such as Bluetooth, this routine is responsible for periodically establishing connections with the slaves and triggering the Level II module to start the session.

4.2 Communication between Level I and Level II modules

As was described earlier, the Level I module is tailored for a specific network medium and operating system whereas the Level II module is common to all platforms. Both Level I and Level II modules are implemented as state machines.

Level I and Level II share data items that serve as the communication area between the two modules. Level II code is always called by the worker routine described above. The Level II state machine is always set in motion by the Level I module. On master-slave type networks (e.g. IrDa), this is done if the Level I module is in the connected state. On peer-to-peer type networks (e.g. UDP/IP) this is done if a data packet destined for the Level II module arrives at the Level I module. In both cases, the Level I module will copy the contents of the packet (minus the headers) to the common area and then run the Level II state machine code. The Level II state machine code will examine the contents of the received packet and change the internal state of the state machine. In addition, if some data needs to be transmitted to the other end, the Level II code will put it into the common communication area. If the Level II code decides that the session is finished, then it sets a specific flag in the common area.

4.3 Level II

Currently, there exists only one version of the Level II code that works on PalmOS and Linux (both user space and kernel space). It may have to be ported to Windows, if such a requirement exists.

The Level II module performs reliable transmission of messages. It provides message segmentation and reassembly, keeps track of lost packets and performs retransmission using the selective acknowledgement (SACK) mechanism. In addition, it provides some rudimentary time synchronization mechanisms along with identification and deletion of old messages.

4.3.1 Level II Sessions

The Level II module consists of a session based protocol. During one session both ends attempt to transmit a single message to each other. Thus at most two messages can be transmitted in one session. However, under certain conditions, more then one session is required to transmit a message. Multiple sessions may be required if the underlying communication medium does not allow for more then two entities to communicate at the same time, thus requiring some type of time division multiplexing. InfraRed and Bluetooth are typical examples of such media. Another case where multiple sessions per message may be required could be bad network conditions that may cause a loss of the control packet. For reasons of time and memory conservation, the CentaurusComm protocol does not provide any mechanism for retransmission of control messages. As is well-known, time and memory considerations are always important in the context of low power devices like PDAs. Therefore, when a packet that carries a control message is lost, the session cannot continue and will hang till a

watchdog timer destroys it. After the session is destroyed by the watchdog timer, a new session is created and the message transmission resumes. Now, because of the way SACKs are implemented, message data that was received in the previous session will not be retransmitted. The Level II module is not responsible for setting up and shutting down of the sessions. All session management must be provided by the Level I module.

4.3.2 Starting transmission

When the Level I module establishes the connection with the peer it resets the state machine of the Level II protocol to the initial state and goes into connected state. In the connected state, every packet that is received by the Level I module is sent to the Level II protocol module. The start up of the session is different for different types of communication media. On media such as InfraRed, one of the devices is selected to be a master. This is the only device that can start a session. The master device is responsible for discovering all the devices in the neighborhood that it can communicate with and polling these devices for messages by establishing a session with each device in a round-robin fashion.

For media that allow multiple nodes to communicate at the same time (either in point-to-multipoint or mulitpoint-to-multipoint mode) the device that has an outgoing message is responsible for establishing the session with recipient. On such devices the Level I module is responsible for maintaining the state of sessions for different devices and loading the correct state for each session. For example, the Level I module for the UDP protocol maintains a session table for each device that opens a session with the local host. Each time a packet is received from a particular device, the Level I module looks up the sender devices in its session table and if an entry exists for the device, it sets up the state machine of Level II module according to the entry in the session table. When the Level II module is done with the processing of the packet the Level I module saves the new state back in the session table. This switching method is not the most efficient but it provides for a reasonable argument against implementing a specialized Level II protocol module for each possible communication medium or having to use multithreading or multiprocessing features of the operating system (if they exist).

When the session is established on peer-to-peer type networks, the Level I module that was responsible for initiating the session sends a 'POLL' message to the local Level II module. When the Level II module receives the POLL message it scans the outgoing objects table and finds the object that should be delivered to the other node. For small mobile devices the table of outgoing objects contains only one entry which is either full or empty, so the selection of the object is trivial. For servers the table will have multiple entries, therefore linear search is used to select the outgoing object. When the object is selected, the Level II module sends an 'OBJ' message that contains the class of the object, it's size and the timestamp, to the other end.

On master-slave type networks the behavior is slightly different - when the Level I module establishes a session, it sends a 'HELO' message instead of 'POLL' to the local Level II module,which then sends the POLL message as a response to 'HELO' message to the other device. This is done as an additional safeguard to prevent the master device from communicating with servers that do not support the CentaurusComm protocol. Since the CentaurusComm protocol was first implemented for master-slave type networks, where latency is relatively low, such a safeguard does not create major overhead. However, on peer-to-peer type networks the latency is very high (500ms) such a safeguard creates too much overhead and therefore it was removed.

Upon reception of an 'OBJ' message the other side examines the class and time stamp on the object and decides to either accept or reject the object. Objects are rejected if either the receiver does not accept the type of object that sender is trying to send or if the receiver already has a newer copy of the object of this type. If the device decides to reject this object it sends back a 'REJ' message. Otherwise it sends back a 'PROCEED' message that contains the bitmap of the already received segments of the object. For the very first session this bitmap has all its bits set to 0.

4.3.3 Message transmission

When the 'REJ' message is received the device removes the object from the outgoing objects table and if this device acts as a slave, it will send a 'NOPE' message to the other side. This will cause sender and receiver to exchange their roles and repeat the session.

When the device receives the 'PROCEED' message it updates it's copy of the bitmap and starts sending packets that correspond to the bitmap entries marked 0. The process of sending packets goes as follows. After it is determined which message segment needs to be sent, the device prepares the packet that whose initial header contains the string 'PK' followed by the slot number and then the contents of the message. This packet is copied to the common communication area and the 'out data' flag is set. After that the Level II state machine switches the state to 'Wait for 'SENT' '. The Level I code picks up this data packet and sends it to the appropriate communication channel.

When the 'PK' message is received by the receiving device it copies the contents to the appropriate slot in the object reception buffer and sets the bit corresponding to this slot to 1.

For peer-to-peer type network media that provide buffering of the outgoing packets, the Level I module sends a 'SENT' message to the local Level II module right after the packet is placed on the network stack to transmit. For master-slave type networks the Level I module places the packet on the network stack and then returns to the event loop of the application. When the network stack completes transmission of the packet it will send a 'packet handled' indication to the Level I code and Level I code will send the 'SENT' message to the local Level II module. When the 'SENT' message is received by the Level II module it sets the bit corresponding to the last transmitted segment to 1. It then checks the bitmap and the object size to determine if it needs to send any additional packets. The same process applies to all messages that are exchanged between two devices, however the synchronous nature of session set up makes the handling of the 'SENT' message optional for the rest of the exchanged messages. If a control message is lost because of the problems with local network stack or during the transmission the whole session will be terminated by the watchdog timer.

4.3.4 Finalizing the session

After the sending device completes sending all the packets it sends the 'DONE' message to the other side and goes to the 'wait for message' state. When the 'DONE' message is received on the receiving side the Level II module checks it's own message bitmap and if all the expected message fragments are received then it responds with the 'ACK' message and updates the timestamp of the last received object in its object acceptance table. It then sends an indication to the application that the message has been received. The application is responsible for processing the received message before it calls the worker routine again, because the contents of the message buffer might get overwritten on the subsequent run of the worker routine.

If the receiving side gets the 'DONE' message and discovers that one or more segments of the message still have their corresponding bits set to 0, it sends a new 'PROCEED' message and a new bitmap to the sender.

Table 1 summarizes all the states of the Level I and Level state machines described in the above sections.

5 Design

5.1 Scenario

A 'SmartRoom' is equipped with a Centaurus Communication Manager, that continuously broadcasts a client application through some medium. A person with a portable device who enters the room for the first time is given the option to install the software. Once the application is installed, it continuously reads the updated list of services. The person is able to choose a service, select a function, fill in the related options and execute the function. These services may be provided by Centaurus systems other than the one the portable device is connected to.

5.2 Components

There are four main components in a Centaurus System: Communication Managers,Service Managers, Services and Clients.

Communication Managers handle all the communication with the Centaurus Client. The Communication Manager is capable of communicating over varied media such as Ethernet, Infra-Red, CDPD and Bluetooth. **Service Managers** are the controllers of the system that coordinate the message passing protocol between Clients and Services. **Services** are objects that offer certain functionality to the Centaurus Client. Services contain information to enable them to locate the closest Service Manager and register themselves with it. Once registered, the Services can be requested by any Client communicating with any Communication Manager. The **Client** provides a user interface for accessing and executing Services.

Figure 1 shows the different components and the relationships between them.

5.2.1 Service Manager

The Service Manager(SM) acts as a mediator between the Services and the Client. When a Service starts up, it has to register with the Service Manager, sending its CCML file. This file contains its name, identification and the interfaces it implements. When a new Client comes along, the Service Manager sends it a ServiceList Object. This ServiceList object changes dynamically, according to the services registered with the Service Manager. So the Client always has the updated

HELO	On master-slave networks causes Level II module to send back HELORESP message and switch from the *idle* state to the *wait* state.
HELORESP	Sent as a response to the HELO message. Not used on peer-to-peer type networks.
OBJ	Used by sender to offer an object to the receiver. Receiver must reply to this message either with PROCEED if the object being offered is acceptable or REJ if it is not.
NOPE	Used on master-slave networks to reverse the roles of receiver and sender. Sent only by slave device in response to POLL message from master.
POLL	Used to query device if it has any objects to send. On master-slave networks, sent by the master (server). On peer-to-peer networks, sent by the Level I module to the local Level II module.
REJ	Sent in response to OBJ message if the receiver does not want to accept the offered object. This happens if either the object type is wrong or an earlier (or newer) object of this type has been already received.
ACK	Sent by receiver in response to the DONE message only when receiver does not expect any more message fragments from the sender. Otherwise, receiver sends PROCEED message again.
PROCEED	Sent in response to the OBJ or DONE messages. This message consists of the message type marker and attached bitmap of received packets. When sender receives this message it updates its own version of the message fragment bitmap and proceeds to send only those message fragments that are not marked as received. When receiver gets this this message it goes into *datasend* state.
PK	Sent by the sender. Each PK message has one message fragment attached to it. When receiver gets this message it copies the fragment to the appropriate slot and then marks the corresponding bit in the message fragment bitmap.
ABOR	Sent by receiver or sender when it wants to abort the current message. The current version of the protocol has very limited support of this message.
DONE	Sent by the sender after it finishes sending the last message fragment. Receiver then determines if all of the message fragments were received it replies back with ACK message. Otherwise it sends back the PROCEED message again.
SENT	A strictly internal message on all types of networks. Sent from the Level I module to Level II module when former detects that the communication stack has finished transmission. Upon reception the Level II module can reuse buffers that were tied up by the transmission process and can begin the transmission of next packet if there is one. Note that on some networks such as UDP this message is sent right after the Level I receives request for transmission from Level II module.

Table 1. Level I and II State Table

list of services. The Client can select a service, which causes the Service Manager to send it the CCML description for that service. The Service Manager then updates its database to reflect that the Client is interested in the Service that is just requested. Whenever the Service Manager gets a status update of the Service, it will send it to all interested Clients. The Client will continue to receive status reports from the Service, until it deregisters itself. The Client sends the new CCML file to the Service Manager, after invoking the interfaces of the Service. On receiving this CCML, the Service Manager validates the Client and the CCML. If the Service is still available, the Service Manager sends the CCML to it, otherwise it is queued for sometime. Once this timeout expires, an error is returned to the Client. The SM is also responsible for service discovery and leasing. It allows Services to register for a certain amount of time. If it does not receive any status update or short 'ping' message (see Section 5.2.3) within that time, the registration is deleted. The SM implements an intelligent lookup for Services, enabling the Clients to search for Services that provide a certain kind of or related function.

In the current design, the main task of the Centaurus Service Manager can be described as the following:

1. Communicate with the Client through CCML
2. Inspect the incoming 'command' or 'update'
3. Dispatch the command to the appropriate services or the Service Management sub-components

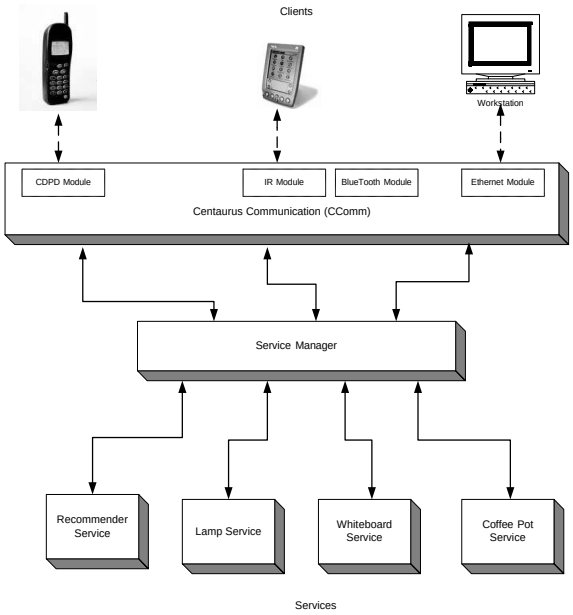


Figure 1. Centaurus Components

4. Handle all status updates, and make sure all interested parties are informed of the updates
5. Interact with Services using CCML
6. Provide service registration services and discovery services

5.2.2 Communication Manager

This is responsible for the communication between the Client and other components of Centaurus. The Communication Manager uses a specific TCP port to communicate with the Service Manager. This is so that Communication Managers and Service Managers need not be on the same system. When the Communication Manager receives information from a Client, it sends this information to the Service Manager. When it receives data from a Service Manager, it validates the data and looks at the header to decide which Client to send it to.

5.2.3 Services

A Service performs a certain action on behalf of the Client. These Services could range from controlling a light switch or a coffee pot to controlling a printer or even a memo pad service, where Clients can leave messages for each other. Each Service registers with a Service Manager by sending its CCML file, along with its name, identification, a brief description of its functionality and its leasing period. Every time its status changes, it informs the Service Manager. If its status has not changed during the leasing period and it wants to renew its lease, it has to send a short 'ping' message to the Service Manager. It accepts requests only from the Service Manager that it is registered with.

5.2.4 Client

A Client is a special kind of Service in that it has to respond to commands and regularly send status updates. A Client talks to the Communication Manager and registers itself with a Service Manager. This registration is similar to the registration of

```
<!--  Entities -->

<!ENTITY % name  "name CDATA #REQUIRED" > <!ENTITY
% value "value CDATA #REQUIRED" >
<!ENTITY % type  "type  CDATA #REQUIRED" >


<!-- Top level element -->

<!ELEMENT ccml
          (system , data?, addons?, interfaces?, info )  >

<!ATTLIST ccml  version CDATA #REQUIRED >


<!-- system declarations -->

<!ELEMENT system    (
             (full, (command|update), valid?, public?,
interactive?,
          id, manager, time, origin, location, parent?, listening? )
              |
             (diff, (command|update), valid?, public?,
interactive?,
          id, time, origin, location,parent?, listening?)
            ) >


<!ELEMENT   command    EMPTY>
<!ELEMENT update EMPTY>
<!ELEMENT full EMPTY>
<!ELEMENT diff EMPTY>
<!ELEMENT valid EMPTY>
<!ELEMENT public EMPTY>
<!ELEMENT interactive EMPTY>
<!ELEMENT id EMPTY>
<!ELEMENT manager EMPTY>

<!ELEMENT time EMPTY>
<!ELEMENT origin EMPTY>
<!ELEMENT location EMPTY>
<!ELEMENT parent EMPTY>
<!ELEMENT listening (id)*>
```

-1-

```
<!ATTLIST id       %name; >
<!ATTLIST manager   %name; >
<!ATTLIST time      %value;>
<!ATTLIST origin    %name; >
<!ATTLIST location  %name; >
<!ATTLIST parent    %name; >


<!--- data declaration -->

<!ELEMENT data (attrib)*>
<!ELEMENT addons  (addon)*>

<!ELEMENT addon  EMPTY>
<!ELEMENT attrib  EMPTY>

<!ELEMENT service (#PCDATA)>

<!ATTLIST addon   %name; >
<!ATTLIST attrib  %name; %type; %value;>


<!--- Interfaces declaration -->

<!ELEMENT interfaces (interface)*>
<!ELEMENT interface  EMPTY>

<!ATTLIST interface  %name; >


<!--- info declaration -->

<!ELEMENT info (description?,icon?) >
<!ELEMENT description (#PCDATA)>
        <!ELEMENT icon (#PCDATA)>
```

-2-

Figure 2. ccml.dtd

Services. On registration, it receives the ServiceList, which contains the current list of Services. The ServiceList is a Service itself, and causes the Service Manager to send the new list of Services, every time a new Service registers, or an existing Service deregisters.

By choosing a Service, the Client expresses interest in it. The Service Manager sends it the CCML file describing the Service. The Client can invoke the specified functions on the Service, by choosing one of its interfaces. After changing values of certain variables, specified in the CCML for the particular Service, it sends the file to the Service Manager to perform that action. It will receive status updates from all Services that it expresses interest in through the Service Manager, until it specifically informs the Service Manager that it no longer wants to receive these messages.

Clients and the Service Managers only exchange CCML messages. By providing XSL transforms, the XML messages can be rendered specifically the Client device. For example, if the Client wants to access a Lamp Control Service and the lamp service can only be turned on and off; this can be displayed as a true/false button on a PDA but on a cell phone the command could be spoken.

5.3 Centaurus Capability Markup Language (CCML)

The CCML is divided into ‘system’ , ‘data’, ‘addons’, ‘interfaces’, and ‘info’, as shown in Figure 2.

The ‘system’ portion contains the header information, the id, timestamp, origin, etc. There are two variables, ‘update’ and ‘command’. A CCML file can only have one or the other. An ‘update’ variable is used to inform other Centaurus components about status updates of Services and Clients, whereas the ‘command’ is only used by Clients to send a command to a certain Service. The system also contains the listening section for a Service or Client. It specifies all the Services that a Service or Client is interested in.

Using the ‘addons’ section, we can add a related Service to another Service, for example, add an Alarm Clock Service to a Lamp-Control Service. We are not currently using this section.

All information regarding the variables and their types are contained in the ‘data’ section.

The CCML for a Client always has one or more ‘actions’ in its data section that a Service Manager can invoke on it. This is used by the SM to change the state of the device.

Actions

- AddService : When this action is set, the Client adds the value of this variable to its InterestList; i.e. the list of services that it is interested in. It is also added to the listening portion of the CCML.
- RemoveService : This is set by the Service Manager, if the Service that the Client is interested in, is no longer available. It causes the Client to stop listening or using the Service and remove the Service from its InterestList. It is removed from the listening portion of the Client's CCML.

The 'interface' section contains information about the interfaces that the object (Service/Client) implements.
Other details like the description, and icon for representation are in the 'info' section.

6 Implementation

6.1 Assumptions

The previous section outlines our overall design, but to facilitate the implementation, we had to make some assumptions and sacrifice some of the features and flexibility. These assumptions in no way compromise the design or results; they only helped in quicker implementation.

The Client tries to find the nearest Communication Manager and register with it. Once registered, the Communication Manager polls the Client regularly for information. This polling completely eliminates the problem of collision, that occurs in a client push method, when more than one Client sends information at the same time. We also have a single Service Manager and two Services for testing. We assume that the client application is installed on the PDA before it enters the 'SmartRoom'. Communication between any two components in the Centaurus System is done via sockets. The Service Manager and the Communication Manager have two dedicated sockets each, one for listening and one for sending information. As the Service Manager and the Communication Manager are at the heart of all communication, we wanted to speed up this process. By giving them a dedicated socket for each type of communication, we reduced the time spent in the creation of a new socket for each connection. Each Service also has a socket for information from the Service Manager. The Service Manager listens to a certain socket for receiving CCML from all the Services. All these sockets are predefined in the Properties file for each component . The information flowing in the system is strictly in the form of CCML.

The Service Manager and the Services have been implemented in Java, whereas we chose C for the IR communication module, Bluetooth communication module, and the Client, for increased efficiency in resource management. We found that most of the service discovery architectures are implemented in Java, like Jini and E-Speak. So, if we decide to move to another service discovery system, integration will be relatively easy as the Service Manager and Services are already in Java.

6.2 Implemented Components

6.2.1 Service Manager

The Service Manager has to listen to two ports, one for incoming messages from Services and one for messages from the Communication Manager. When a Service registers itself, the Service Manager adds it to its list of Services by recording the CCML and the port number the Service listens to. It also starts a timeout for the leasing period of the Service. If it does not receive an update or a 'ping' within the timeout, it removes the Service from its list. Every new Client is added to the Clients list. All the Services it is interested in are added to the Service-Client list with both the Service and Client IDs. Whenever the Service Manager receives an update from a Service, it updates its Services list and reads its Service-Client list and sends the new CCML to every Client in that list. When a Client sends an update, the Service Manager changes the Clients list. It then reads the list of Services that the Client is now interested in, and appropriately modifies the Service-Client list.

6.2.2 Client

The Client attempts to discover and communicate with the Communication Manager through a certain medium. Once discovered, it registers with the Centaurus system. It maintains a list of Services that it is interested in, called InterestList. Whenever a Client receives a command from the Service Manager, its checks the action specified, AddService or RemoveService, and adds or removes Services accordingly from its InterestList. If it receives an update, it checks if the Service is in its InterestList. If the Service is in its InterestList, the Client renders the CCML and waits for user input. If the Service is not in its

InterestList, the Client discards the message. If the user changes any variables of the Service's interface, the Client modifies the Service's CCML and sends it to the Communication Manager, which in turns forwards the command to the Service Manager. The Service Manager makes sure that the CCML is sent to the appropriate Service.

6.2.3 IR, CDPD and Bluetooth Communication Modules

Communication modules that handle communication via IR, CDPD and Bluetooth have been implemented. These modules form Level II of the CentaurusComm protocol described in Section 4. These modules are used both by the Communication Manager as well as the Client. However, client functionality is quite different from Manager functionality.

The IR module for the Communication Manager listens on a fixed port for updates from the Service Manager. It implements a complicated InfraRed protocol. An enhanced and modified Linux IrDA stack forms the core of the IR module. Modifications to IrDA stack were necessary for better handling of disconnections and discovery. The module establishes and maintains two TCP/IP connections with Service Manager and communicates via IR with mobile Clients. It listens to both sides for incoming CCML messages and transmits them to the appropriate destination. The Client version of the module has been optimized for the IrDA stack on PalmOS. The module uses the built-in IR port on the Palm Pilot for communication.

The CDPD module for the Communication Manager has been implemented by combining it with the UDP protocol. Thus, the message objects are sent and received as UDP messages with the Level I of CentaurusComm providing reliability. The Manager listens for messages from the Client on a fixed UDP port and on a fixed TCP port for messages from the Service Manager. The Client version of the CDPD module has been highly optimized for Palm Pilots. In a manner similar to the IR module, it attempts to discover the Communication Manager by sending discovery messages to the fixed UDP port. One very important feature of this module is that it does not attempt to transmit until it determines that the signal from the CDPD modem to the base station is stable. This test for signal stability is performed before every attempt at transmission.

The Bluetooth module for the Communication Manager and Client is a preliminary version that also uses UDP for sending and receiving CCML. The Communication Manager version initializes the Bluetooth hardware, sets up the Bluetooth protocol stack and starts up the PPP daemon. It also starts up the Bluetooth Service Discovery Protocol (SDP) server. The Client version is slightly different. It first initializes the Bluetooth hardware and sets up the Bluetooth protocol stack. It then begins device discovery. On discovering the Bluetooth device that the Communication Manager is using, it establishes a PPP connection with the stack after appropriate service discovery. It then attempts to discover the Manager and establish a session with it. Once the session is established, it is ready to send and receive CCML messages.

6.2.4 Services

We have developed one hardware related service for controlling a lamp and one software service for playing MP3 files. There is another Service, ServiceList, that is an inherent part of the protocol, and is used for providing an updated list of services to the Client.

We have implemented a Service class and ServiceInterface class that handle validation of the CCML, the registering of the Service with the Service Manager and the sending of the updates. All Services implemented in Java should, for conformity, extend the Service class, and implement the ServiceInterface class. The ServiceInterface class contains a commandHandler function that has to be implemented by every Service that implements the interface. This is the function that handles changes to the CCML file of the Service. A Java Service need only implement a constructor and this commandHandler to be integrated into a Centaurus system.

As mentioned earlier, the Centaurus system also handles non-Java Services as long they can use CCML and either communicate via sockets with the Service Manager or with a Communication Manager through some native protocol.

- ServiceList

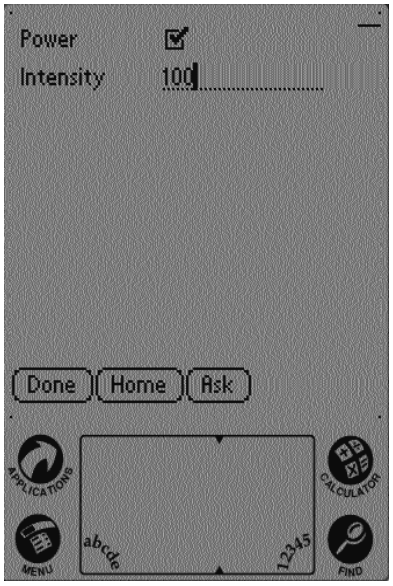
  Each time, a Service registers or is no longer available, the ServiceList triggers the Service Manager to send the updated list of Services to all the Clients. This does not use the Service class or the ServiceInterface class. It is contained completely in the Service Manager. It is a special Service because it is handled in same way as other Services are, but within the Service Manager itself. Figure 3(a) shows a screen shot of the ServiceList received by the Client (a Palm Pilot).

- Lamp-Control

  Using X10 [15] devices and FireCracker [14], we were able to control a lamp in the room. We can extend this to control any device because X10 is a power-line carrier protocol that allows compatible devices to communicate with each other


MP3 Player
Lamp
Done
Home
Ask
APPLICATIONS
CALCULATOR
MENU
FIND
abcde
12345

(a)


Power
Intensity
100
Done
Home
Ask
APPLICATIONS
CALCULATOR
MENU
FIND
abcde
12345

(b)


Lamp
MP3 Player
Done
Home
Ask
APPLICATIONS
CALCULATOR
MENU
FIND
abcde
12345

(c)


las-vegas
cruising
lady
desert-rose
higher-love
fine-night
wasnt-me
rotterdam
thank-you
Done
Home
Ask
APPLICATIONS
CALCULATOR
MENU
FIND
abcde
12345

(d)


las-vegas
cruising
lady
desert-rose
higher-love
fine-night
wasnt-me
rotterdam
thank-you
Done
Home
Ask
APPLICATIONS
CALCULATOR
MENU
FIND
abcde
12345

(e)

via the existing 110V wiring. FireCracker is a Java class that allows a computer to communicate with the X10 device. The Service constructor makes sure that the X-10 device works. The commandHandler function looks for the value of the interfaces. If the 'Powered' interface has a value that is different from the status of the Power variable, then the commandHandler proceeds, otherwise the command is discarded. If the value is true, the lamp is set on, otherwise the lamp is set off. The CCML file is changed and an 'update' is sent to the Service Manager. Figure 3(b) shows a screen shot of the Client using the Lamp Service.

Figure 3(c) shows that the order of the services has changed after the Client selected one of them. Thus, all clients listening to these services will know that some other client is currently using one of services.

- MP3 Player

  Figure 3(d) shows a shot of the list of songs in MP3 format provided by the MP3 Service. In figure 3(e) we see that the Client has selected 3 songs it wants the MP3 Player to play.

  We are using a popular MP3 player for Unix, mpg123 [18], that has a Java wrapper around it to allow us to plug it into the rest of the system. The constructor for the Service, reads all the .mp3 files from a specified directory and creates its CCML files. It has a number of CCML interfaces, one for each song it can play. The commandHandler function checks the CCML interface and reads the songs selected. These songs are checked against the current list of songs. If they are valid, they are fed into mpg123 [18]. The new CCML file is created and sent to the Service Manager.

6.3 Functions

6.3.1 Registration

Both Services and Clients have to register with the Centaurus system to be visible.

- Services

  A Service on starting up, reads its properties file and retrieves the Service Manager's port number. After creating its CCML file, it sends its CCML and the port number that it is listening to, to the Service Manager's Service port. The Service Manager validates the CCML and adds the Service to its Services list.

- Client

  When a PDA enters the 'SmartRoom', we assume that it has the Client application installed on it, as mentioned in the beginning of this section. It locates the Communication Manager and sends its pre-defined CCML file. The Communication Manager sends this CCML to the Service Manager, after validating the CCML checks if the Client already exists in its Clients list. If it does, then the Service Manager updates its list, otherwise it adds the Client to its list of clients. The Service Manager adds the ServiceList action and sends the CCML back to the Client. The Client adds the ServiceList to its listening section and sends the CCML back to the Service Manager, prompting the Service Manager to send it a list of currently registered Services.

6.3.2 Using the Service List

The ServiceList is used to provide an updated list of Services to the Clients. When the Service Manager gets the CCML of a new Client, it sets the AddService action in data section of the CCML to ServiceList.

$< data >$
$< attribname = "AddService" type = "action" value = "ServiceList" / >$
$< /data >$

It also sets the 'command' variable in the header.

This new CCML is sent back to the Client. The Client realizes that it is a command and checks the actions. The Client then adds the ServiceList to its currently empty list, InterestList which is the list of Services that the Client is interested in.

$< listening >$
$< idname = "ServiceList" / >$
$< /listening >$

When the Client is polled next by the Communication Manager, it sends its updated CCML. The Service Manager reads the listening section, and finds the ServiceList. It updates its Service-Client list and sends the list of Services in CCML to the

Client. Whenever the list of Services changes, the Service Manager goes through its Service-Client list and sends the new list to all the Clients that are interested in the ServiceList. In this way, the ServiceList works like any other Service, except that it is contained within the Service Manager.

If both our Services have registered then the data portion of the ServiceList would look like

$< data >$
$< attribname = "MP3Player1.0" type = "bool" value = "false" / >$
$< attribname = "Lamp - 001" type = "bool" value = "false" / >$
$< /data >$

6.3.3 Requesting a Service

When a Client receives the list of Services, it displays this list for the user. The user can select a Service to use. The Client then creates a command for the ServiceList. It changes the data portion of the ServiceList, with the value of the Service selected as 'true'. This is sent back to the Service Manager. As it is a command for the ServiceList, which is part of the Service Manager, the Service Manager handles it. From the system section, the Service Manager retrieves the name of the Client and checks the data section for the Services. It then retrieves the latest CCML for the Client from its Clients list and creates a command for the Client. It sets the AddService action to the Services selected and sends the CCML back to the Client. The Client processes this CCML as it would any AddService action by adding the Service to its listening section. It also adds the Service to its InterestList. When the Client is next polled it sends its updated CCML.

From the updated CCML, the Service Manager reads the list of Services that the Client is listening to, and picks out the new Services that the Client was not previously listening to. It sends their CCML to the Client via the Communication Manager. It then, adds the new Service-Client pair to its Service-Client list.

Once the Client gets the CCML of the Service, it displays it for the user. The user can use the interfaces to perform actions. The Client modifies the Service's CCML to make a command, sets the new values and sends when polled.

The Service Manager realizes that it is a command and sends it to the appropriate Service. The Service carries out the command and sends the update to the Service Manager, which propogates the update back to the Clien.

6.3.4 Status Update

If a Service Manager receives an update from a Service, it checks its Service-Client list for all the Clients interested in this Service. It sends the updated CCML to these Clients.

When a Service Manager receives an update from a Client, it carries out certain functions on it. It checks the listening section and retrieves the list of Services that the Client is listening to. It picks out the new Services, ones that the Client was not previously listening to. It sends their CCML to the Client via the Communication Manager. Then for each new Service, it adds a new Service-Client pair to its Service-Client list.

7 Protocol

7.1 Overview

In this subsection, we give a brief overview of the entire Centaurus protocol detailing the data flow within the system. We also explain the control flow from one component to another in various situations.

- On entering a room, the Client tries to locate a Communication Manager that it can communicate with.
- Once a new Client registers, the Service Manager is contacted. The CCML file for the Client is sent to the Service Manager through a pre-defined port number. The Service Manager creates a new ID and sets the action, AddService to ServiceList in the CCML file for the Client and returns this CCML to the Communication Manager.
- Whenever the Communication Manager receives information on its reading port, it checks the header to find the Client id and then forwards it to the correct Client.
- The Communication Manager polls all the Clients in the room at regular intervals. A Client responds by sending its own CCML as an 'update' or sending the CCML of a service it wants to use by issuing a 'command' .

- When the Service Manager receives any CCML from a Client, the CCML is first validated and the Service Manager makes sure that it came from an authorized Client. Then it checks the header to decide whether the CCML is an 'update' or a 'command'.
- If it is an 'update', it reads the list of Services that the Client is listening to, and picks out the new Services that the Client was not previously listening to. It sends their CCML to the Client via the Communication Manager. It then, adds the Service-Client pair to its InterestList.
- On the other hand, if it is a 'command', it extracts the name of the Service, and sends it to the appropriate Service.
- If a Client receives an 'update', it is the status update of a Service that the Client is interested in.
- If a Client receives a 'command', it knows that the Service Manager has set an action on it. Following this, the Client validates the CCML, and if there is any AddService action, adds the value to the listening list in its CCML. It also sets the 'update' variable in its CCML header. If the action is RemoveService, it removes the Service from its list.
- After starting up, a Service has to register with the Service Manager, by sending its CCML file and its port number to the Service Manager. The 'update' variable is always set in a Service's CCML.
- When a Service registers, the Service Manager enters its information into a table and sets a timeout. If the Service does not send an update without the timeout, it is deleted from the table.
- When a Service receives a 'command' from a Service Manager, it tries to carry out the command. It then updates its CCML to reflect the new changes and sends it to the Service Manager.
- Whenever a Service Manager, receives an 'update' from a Service, it forwards this update to all Clients in the Service-Client list that are interested in this particular Service.
- If the Service has not changed its state during the lease period and wants to renew its lease, it sends a short 'ping' message to the Service Manager.
- On receiving a 'ping' message, the Service Manager renews the lease of the associated Service
- If the lease period of a Service expires and the Service Manager has not got a status update or ping from the Service, the Service is automatically de-registered.

8 Experiments with Centaurus using IR, CDPD and Bluetooth

The Service Manager and Services were initialized and run on a Pentium II based system running Linux 2.2.14 connected to a 100BaseT network. This system also directly controlled the Lamp and executed the MP3 Player whenever requested by the Client. The wireless interface devices - IrDA and Bluetooth - were connected on serial ports to another Pentium II based system running Linux 2.2.17. This system was also on the 100BaseT network. The Communication Manager was started up on this system after being integrated with the required communication module (IR, CDPD or Bluetooth) based on the medium being used for the experiments.

In order to experiment with Centaurus using IR, multiple PDAs running an instance of the Client over CentaurusComm were brought within line-of-sight of the IR dongle connected to the Linux system. As expected, the Client on each PDA was able to establish a session with the Communication Manager. A form containing the list of services (Lamp and MP3 Player) was then displayed by the PDAs. The services were successfully manipulated (turning on/off the Lamp and getting the MP3 Player to play a list of specified songs) from the PDAs.

An OmniSky CDPD modem was connected to each PDA used in the experiment. CDPD is a technology that supports IP type networks. Therefore, unlike IR, no line-of-sight requirements exist. The modem is able to communicate with a base station (cell tower), operated by the service provider, wirelessly. The base station communicates with the Linux system over the wired network. While performing the experiments, we noted that as long as the signal on the modem was unstable, the Client did not attempt to establish a session with the Communication Manager. In a dynamic environment, multiple PDAs can attempt to manipulate the same service. Therefore, it is necessary that as a particular service is being used by a particular PDA, other PDAs interested in the same service be notified of this fact. It is the responsibility of the Service Manager to handle these issues. Our experiments showed that the Service Manager does indeed transmit updates to all PDAs whenever

required. In one experiment, we enabled two PDAs to subscribe to the Lamp service and allowed one of them to turn the lamp on. The Service Manager immediately sent an update to the other PDA indicating the new status of the Lamp service.

The Client using the Bluetooth communication module was set up on Pentium II based laptops running Linux 2.2.18. Each Bluetooth device was attached to the serial port of a laptop. As soon as the Client was started up discovery discovery and the following processes were executed as described in 6. Experiments with Bluetooth as the medium were also successful.

Bluetooth is a new technology and expectations that it will become ubiquitous are high. In order to place this technology in perspective, we have evaluated CentaurusComm and TCP performance over Bluetooth. We executed a client program on an Intel Pentium running Linux 2.2.18 using the Bluetooth stack developed by IBM, called BlueDrekar. We used Bluetooth hardware developed by Ericsson. A concurrent server executed on an Intel Pentium running Linux 2.2.17. In all experiments, the wireless client initiates the connection to the server. All experiments were done on a per-session basis. A session consisted of a request message from the client, a reply message from the server and a good bye message from the client. The good bye message consisted of all recorded values on the client, like Round Trip Time (RTT), number packets transmitted and received and number of bytes transmitted and received. Experiments consisted of 1000 sessions with variable packet sizes of 64, 128, 1024, 2048, 4096 and 8192 bytes. We chose RTT as the primary performance metric. We analyzed the performance of both TCP and CentaurusComm over Bluetooth based on the results of the experiments.

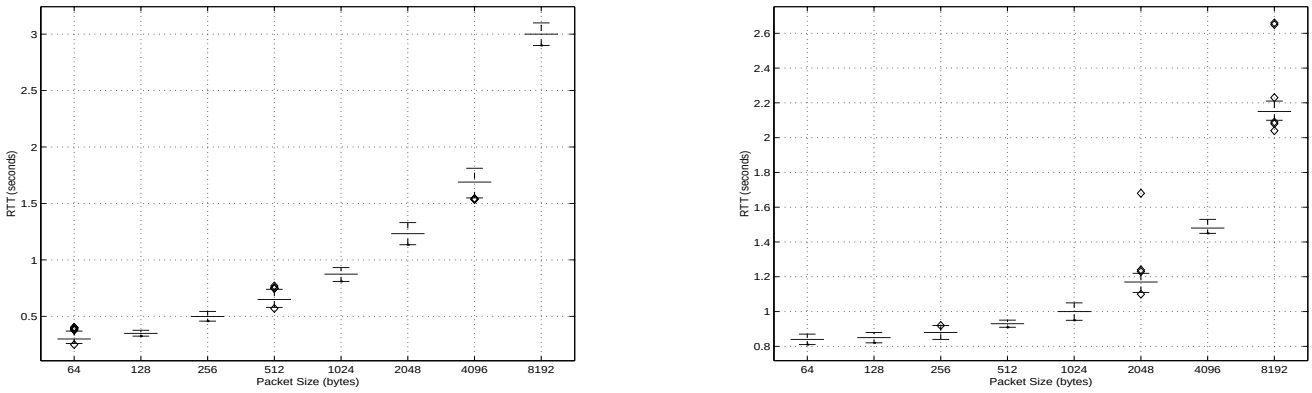


Figure 4. (a) Performance of TCP over Bluetooth (b) Performance of CentaurusComm over Bluetooth

The graphs in Figure 4 show performance TCP and CentaurusComm for different packet sizes. We find that for small packet sizes (64 to 512 bytes), CentaurusComm performs almost constantly (0.8 to 0.9s), whereas TCP performs almost linearly (0.25 to 0.65s). For packet sizes 1024 bytes and above, both protocols perform linearly. We note that the RTT for TCP is lower that of CentaurusComm for packet sizes upto 1024 bytes because TCP has the advantage of running in kernel space. However, for larger packet sizes we note that CentaurusComm performs better than TCP.

9 Conclusions and Future Work

We have successfully developed the first version of Centaurus. We believe that our infrastructure is appropriate and effective for deploying services in an indoor environment. Such environments are typified by handheld clients connecting to services on the fixed infrastructure using wireless ad-hoc networks such as those based on Bluetooth. The first stage development, including the Service Manager, Communication Manager, MP3 player services, Lamp services etc. have verified that our vision of dynamic service discovery and management over ad-hoc wireless networks is feasible.

One of the important lessons that emerges from this design is that cross layer interactions allow for more efficient systems. For instance in Centaurus COMM protocol, we can exploit the fact that services are exchanging short messages to create a transport protocol that is far more robust and efficient compared to TCP on wireless media such as CDPD and Bluetooth. Similarly, routing of packets in the ad-hoc network is best done based on service names. That is, a message is addressed to a particular service, and the service manager can deliver it appropriately. This is far more efficient than trying to use IP based routing when many IP services like name resolution may not be available in an ad-hoc network. Both of these are

examples where there are cross layer interactions between networking and service discovery / data exchange parts of the system. Similar cross layer interactions are exemplified by the exchanged data describing the interface to the service so as to allow dynamic interface generation by the clients.

We are building on this basic framework by adding attractive interfaces for the portable devices, creating new services, enabling more intelligent brokering of services, and adding mechanisms to support privacy and security. We briefly describe our ongoing work on these in the next few paragraphs. We are also transitioning our system to work in situations where the services themselves come from devices on the ad-hoc network.

9.1 Security

We are currently working on the security infrastructure. We are following a Kerberos-like [16, 17] system and have a ticket granting server (TS). Each Service Manager on starting up will register with the TS. Every mobile device that wants to use Centaurus, has to go to this TS and submit its CCML file. From the CCML and checking its policies, the TS decides what Service Managers it can access.

The ticket is generates as follows

1. The rights that the device can possess are encoded into a Prolog[19] like clauses as follows

   ```
   canUse(ServiceManager12)
   canUse(ServiceManager2)
   ```

2. It also contains a clause that says how long the ticket is valid

   valid(1105001130, 1110005067)

   This means that the ticket is valid from 1105001130 to 1110005067.

3. These rules are signed with the TS's private key to create a signed message.

4. The TS calculates the checksum of the system section of the device's CCML of the device and the signed message. This acts as the ticket.

This ticket is sent back to the device.

When the client needs to access a Service on a certain Service Manager, it sends its CCML and the ticket. The Service Manager will check that the ticket is valid and from the TS. If the ticket is valid, the Service Manager makes sure that the device has permission to access its Services by checking the rules in the tickets. We are still contemplating where the Service permissions should be set. Either the Service Manager can decide which Services the client should be able to access and change the ServiceList so that the client can only see the Services that it can access or the Service Manager does nothing and passes the ticket to the Service selected by the device, and the Service decides whether to allow the access.

9.2 Recommender Service

We are also working on a Recommender Service. Instead of returning a list of all possible services that are available to a Client, this service recommends a list of services that might be in the interest of the Client based on the existing environment context. For example, the system returns a coffee-maker control service during the morning to the user, and in the evening it returns a light control service to the user. It may also notice that the user generally wants to listen to to the same list of songs and provide the list as soon as the user steps into the room.

9.3 Service Managers Hierarchy

We would like to arrange the Service Managers into a hierarchy so that the Services could connect to the closest Service Manager, and the location of a Service Manager need not be coded into the Services. This will also allow the Services to be shared across the Service Managers, so a user could enter one room and use the printer in another room by using the printer Service on the Service Manager in the other room.

9.4 DAML

Most of the information on the web today is described in HTML which is not very powerful or expressive. The World Wide Consortium (W3C) developed XML to help describe information better. As a step towards the Semantic Web, DARPA Agent Markup Language (DAML [26]) is an effort to allow the use of ontologies to describe objects and their relationships to other objects.

We are working to replace CCML with DAML [26] because of its expressivity and its effectiveness in constructing ontologies. DAML's ability to include rules will allow us to increase the intelligence of the system. Clients and services will be able to define their own axioms and rules that the system can incorporate, which will help the system serve both clients and services better.

References

[1] The Ninja Project **http://ninja.cs.berkeley.edu/**

[2] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz, "An Architecture for a Secure Service Discovery Service" Fifth Annual International Conference on Mobile Computing and Networks (MobiCom '99)

[3] Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber, and Lawrence Rowe, "Composable Ad hoc Mobile Services for Universal Interaction", In Proceedings of the 3rd ACM/IEEE MobiCom, Budapest, Hungary, Sep 1997, **http://bmrc.berkeley.edu/research/publications/1997/145/thodes.html**

[4] T. Hodes, R. H. Katz, "A Document-based Framework for Internet Application Control", Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99), October 1999

[5] Manuel Roman and Roy H. Campbell, "Gaia: Enabling Active Spaces", In Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark, September 2000.

[6] Fabio Kon, Christopher K. Hess, Manuel Roman, Roy H. Campbell, and M. Dennis Mickuna, "A Flexible, Interoperable Framework for Active Spaces", OOPSLA'2000 Workshop on Pervasive Computing. Minneapolis, MN, October 16, 2000.

[7] W3C Consortium **http://www.w3.org/XML/**

[8] The Official Bluetooth Website **http://www.bluetooth.com/**

[9] Internet Business Solutions : E-Speak
http://www.e-speak.hp.com?qt=espeak/

[10] Jini Connection Technology Executive Overview
http://www.sun.com/jini/

[11] Service Location Protocol
http://www.svrloc.org/index.html

[12] Infrared Data Association **http://www.irda.org**

[13] Mark Taylor, William Waung, Mohsen Banan, "Internetwork Mobility: The CDPD Approach", Prentice Hall Professional Technical Reference, September 1996

[14] FireCracker **http://www.x10.com/welcome/firecracker/**

[15] X10 Devices **http://www.x10.com/**

[16] B. Clifford Neuman and Theodore Ts'o., "Kerberos: An Authentication Service for Computer Networks", IEEE Communications, September 1994

[17] John T. Kohl, B. Clifford Neuman, and Theodore Y. T'so, "The Evolution of the Kerberos Authentication System", Distributed Open Systems, IEEE Computer Society Press, 1994 **ftp://athena-dist.mit.edu/pub/kerberos/**

[18] Mpg123, MP3 Player for Linux/Unix Systems **http://mpg123.org/**

[19] Swedish Institute of Computer Science, SICStus Prolog **http://www.sics.se/sicstus/docs/latest/html/sicstus.html**

[20] University of Washington, Dept of Computer Science and Engineering, "Portolano: An Expedition into Invisible Computing" **http://portolano.cs.washington.edu/**

[21] "The Invisible Computer", D. Norman, MIT Press, 1998.

[22] Stanford Interactive Workspaces Project
http://graphics.stanford.edu/projects/iwork/

[23] Monarch Project **http://www.monarch.cs.cmu.edu/**

[24] M.Blaze, J.Feigenbaum, J.Lacy, "Decentralized Trust Management" ,IEEE Proceedings of the 17th Symposium on Security and Privacy, 1996

[25] M.Blaze, J.Feigenbaum, M.Stauss, "Compliance Checking in the Policy Maker Trust Management System", Proceedings of Financial Crypto'98, Lecture Notes in Computer Sciences vol.1465, Springer Berlin, 1998

[26] The DARPA Agent Markup Language Homepage **http://www.daml.org/**