Wireless Networks 8, 619–635, 2002

Centaurus: An Infrastructure for Service Management in Ubiquitous Computing Environments

LALANA KAGAL *, VLADIMIR KOROLEV, SASIKANTH AVANCHA, ANUPAM JOSHI, TIM FININ and YELENA YESHA
Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, USA

Abstract. In the near future, we will see dramatic changes in computing and networking hardware. A large number of devices (e.g., phones, PDAs, even small household appliances) will become computationally enabled. Micro/nano sensors will be widely embedded in most engineered artifacts, from the clothes we wear to the roads we drive on. All of these devices will be (wirelessly) networked using Bluetooth, IEEE 802.15 or IEEE 802.11 for short range connectivity creating pervasive environments. In this age where a large number of 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 user. This paper outlines the design of Centaurus, an infrastructure for presenting services to heterogeneous mobile clients in a physical space via some short range wireless links. The infrastructure is communication medium independent; we have implemented the system over Bluetooth, CDPD and Infrared, three well-known wireless technologies. All the components in our model use a language based on Extensible Markup Language (XML) for communication, giving the system a uniform and easily adaptable interface. Centaurus defines a uniform infrastructure for heterogeneous services, both hardware and software, to be made available to diverse mobile users within a confined space.

Keywords: mobile computing, service management, ubiquitous computing, pervasive computing

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 [16,20]. The "SmartHome" and "SmartOffice" scenarios present a step towards realizing this vision. 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" will use sensors to gather information about the user and environment, and allow the user to use more interactive forms of input like voice, eye movements etc. 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 heterogeneous 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 automatically become aware of each other and can exchange information to cooperatively 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 management. Within a confined space, the client can access services provided by the nearest Centaurus System via some short-range wireless technology. Centaurus 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 communicate via Centaurus Communication Markup Language (CCML) (described in section 5) which is based on Extensible Markup Language (XML[1]). We found that this W3C Standard is very useful in describing ontologies, and defining properties and interfaces of services. It will also help in integrating Centaurus with emerging semantic languages like DARPA Agent Markup Language (DAML + OIL) [10]. The Communication Manager is flexible and allows any medium to be used for communication, but for implementation purposes, we have used

* Corresponding author.

[1] `http://www.w3.org/XML`

Infrared [11], CDPD [18] and Bluetooth [4]. 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 work in the area of "smart environments", and compares Centaurus with them. Section 3 describes the features of the system with an example. The design of Centaurus is outlined in section 4. The Centaurus Capability Markup Language is described in section 5. Implementation details are sketched out in section 6 and certain important functionality of Centaurus is discussed in section 7. Some experiments with the Centaurus system are described in section 8. We present some possible directions of our future research work in section 9 and conclude with a summary in section 10.

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 [6], the Portolano project [5] at the University of Washington, Stanford's Interactive Workspaces Project [17], Berkeley's Document-Based Framework for Internet Application Control [8,9] and Active Spaces [12,15] at the 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. They have also designed ICrafter, a service framework for these workspaces that generates user interfaces of groups of services for requesting clients [14].

University of Washington's Portolano project is in the early stages and is involved in "invisible computing" a term invented by Donald Norman [13] 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 [12,15] 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, as long as they use CCML to communicate. 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 Ethernet. Centaurus has been specifically designed 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 [6]. 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 an approach is the decreased complexity of distributed state management and increased fault tolerance. Even in the event of a 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. For security and information assurance, Ninja utilizes encryption between all entities within the system. This implies a high computational overhead on the endpoints of the communication regardless of whether the endpoint is a PDA, cell phone, or a powerful workstation. We are incorporating a simplified Public Key Infrastructure and distributed trust [2,3] principles to provide a security infrastructure for Centaurus [19]. Centaurus does not make the assumption that the end points are computationally robust. The entities in the Centaurus system will enjoy non-repudiation, authentication, and protection from replay attacks via the simplified PKI. SmartSpaces lack central control that necessitates the use of distributed trust, which will also provide more flexibility than traditional security mechanisms. Moreover, while Centaurus is protocol and communications medium independent, Ninja is not. 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. We also do not distinguish between hardware and software services, allowing the user to use either in the same way. Since all communication between services and clients in Centaurus is 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 "smart homes" and "smart offices". This is because of its independence from 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 (described in section 4.3) in the event of the Service Manager failure.

3. A use case for Centaurus

Consider John walking into a meeting room with his PDA phone. He is late and the meeting has already started, but it is 8:00 AM and he needs a cup of coffee. He looks around for a coffee maker, but cannot see one. In the meantime the Centaurus client on John's PDA registers itself with the Centaurus system in the room by sending its CCML description. The Centaurus system sends the client a list of currently registered services in CCML. The CCML is rendered by the client in a form appropriate to the device. This allows heterogeneous devices to use Centaurus without any modification to the client or the interface. John looks at the list and seeing a coffee maker selects it. The client on his PDA reads this input and sends it back to the Centaurus system. The Centaurus system notes John's need to use the coffee maker and sends John's PDA a CCML description of the coffee maker and its interface. The description includes the location of the coffee maker, which is in the next room. The interface is rendered by the client and John enters '1' in the text box marked *Number of Cups* and this message is sent back to Centaurus. Centaurus validates this message and then sends it to the coffee maker service. The coffee maker service parses the CCML to retrieve the command and executes it. The execution causes the coffee maker to change its state, and so it sends a status update to Centaurus. Centaurus knows that John's PDA is trying to use the coffee maker, so it sends the status update to John's PDA as well. John gets an instant update of the coffee maker and knows that his coffee is being brewed. When the coffee is brewed, John is sent a message and he walks over to the next room to pick it up.

4. System design

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.

Centaurus consists of several components: the CentaurusComm Transport Protocol, Communication Managers, Service Managers, Services, and Clients. CentaurusComm Transport Protocol is an efficient, message-object based transport protocol which abstracts out the medium specific information. Communication Managers handle all the communication with the Centaurus client using different modules of CentaurusComm protocol. The Communication Manager is capable of communicating over varied media such as Ethernet, Infrared, CDPD and Bluetooth. Service Managers control access to the services and act as gateways between the services and clients. Services are objects that offer certain functionality to Centaurus clients. 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 an interface to the user to interact with the services.

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

4.1. The transport protocol in Centaurus: CentaurusComm

4.1.1. System architecture

Figure 2 shows the design of the CentaurusComm architecture from the perspective of data exchanges. Figure 3 shows the interaction of the components in the architecture based on the exchange of control messages.

CentaurusComm consists of one or more lower level protocol modules (designated as Level I), one higher level module (designated as Level II) and an application program interface. Level I modules are communication medium dependent; the Level II module is medium independent. The API is responsible for accepting the objects from the application layer for transmission and notifying it when messages are received.

The protocol is implemented as a collection of data structures and state machines. The principal data structures include the *transmit* queue and the *receive* queue. 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. This is in contrast to TCP, which requires substantial support from the OS for signaling. CentaurusComm is designed such that the the transmission, reception and recovery procedures are divided into many small subprocedures that last for a very short time. With the exception of domain name resolution, which occurs very infrequently, the protocol never blocks. This design gives an impression of concurrent execution of the user program and the protocol modules.

The Level I and Level II protocol modules share data items in order to communicate with each other. The Level I module copies the contents of any received data packet (excluding the headers) to the common area and runs the Level II state machine. The Level II state machine examines the contents of the received packet, based on which, it changes state. If, based on the new state, a response is in order, the Level II module places the response in the common communication area. In addition, if the Level II state indicates that the session is finished, the Level II module sets a specific flag in the common area.

4.1.2. The Level I module

The principal driver of the CentaurusComm protocol is a *worker routine* that is part of the Level I module(s). On startup, the user application must always call this routine. The main purpose of the *worker routine* is to perform message transmission and handle message reception. Based on the status of the *transmit* and *receive* queues, the network connections and the Level II state machine, the worker routine takes one of the following actions:

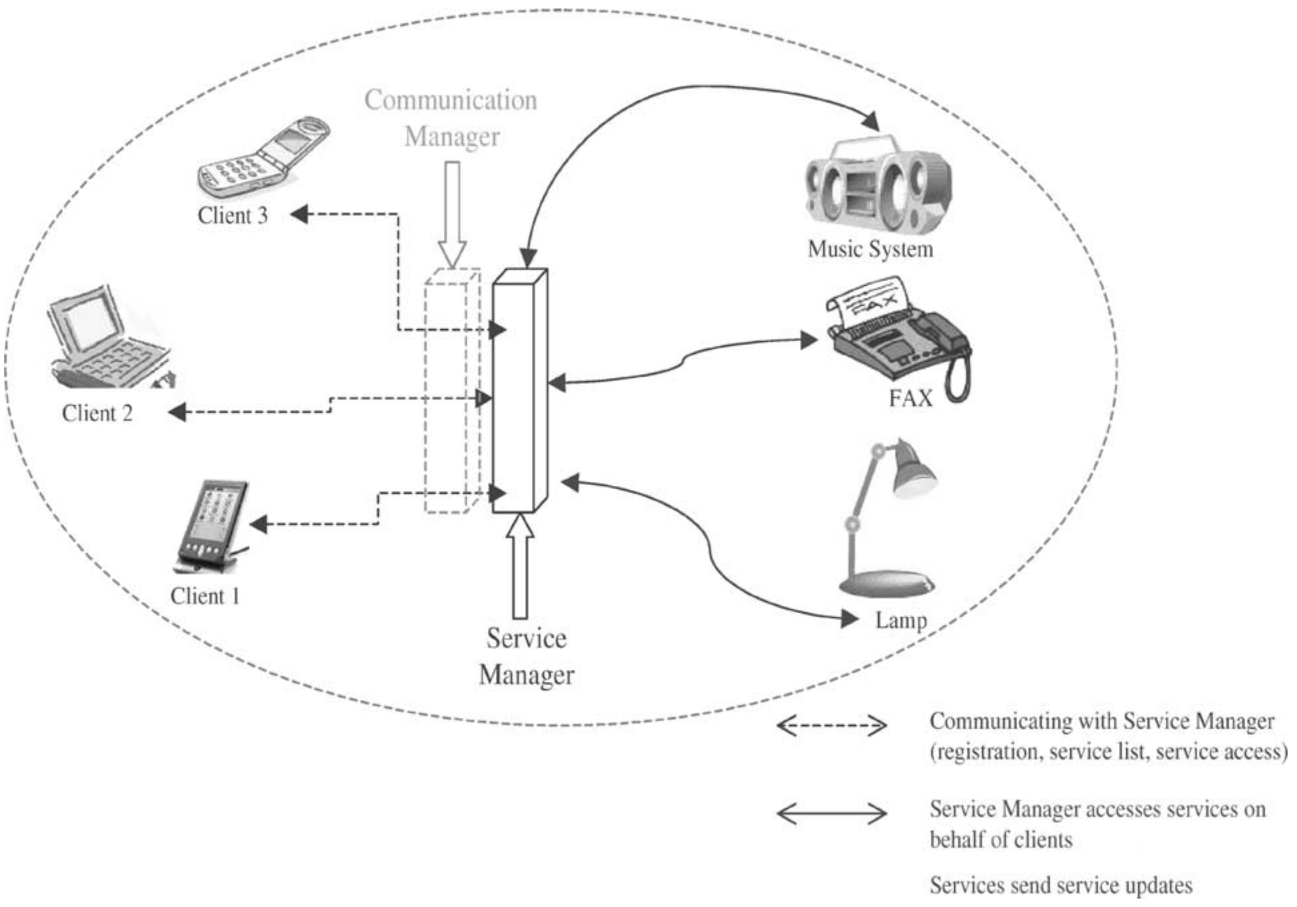
Communication Manager
Client 3
Client 2
Client 1
Service Manager
Music System
FAX
Lamp
Communicating with Service Manager (registration, service list, service access)
Service Manager accesses services on behalf of clients
Services send service updates

Figure 1. Components of Centaurus.

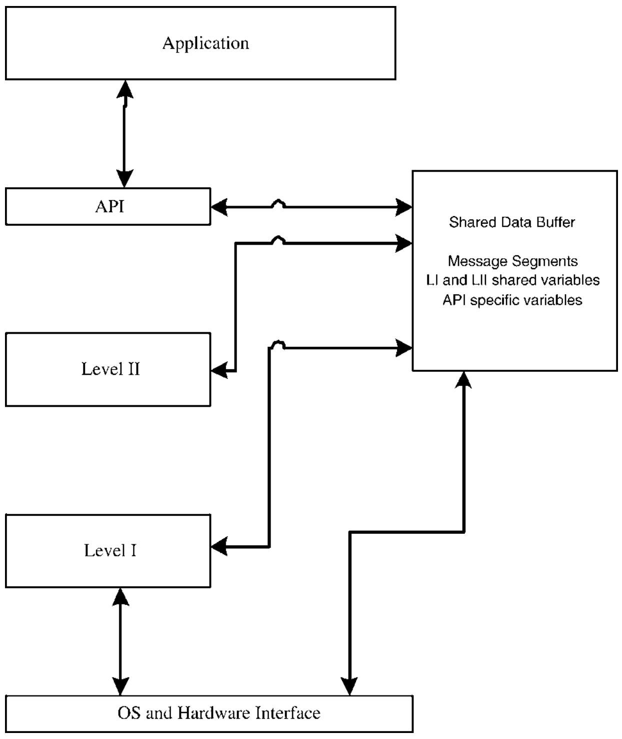

Figure 2. Interaction between components of CentaurusComm protocol (data).

- If data is present in the *transmit* queue, schedule it for transmission.
- If data is present in the *receive* queue, run the Level II state machine in order to process it.
- In peer-to-peer type networks, examine the table of outgoing messages and trigger the Level II state machine in order to start transmission.
- In IrDA-based networks, cause the “master” to periodically establish connections with the “slaves” and trigger the Level II state machine to start a session.

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 As long as the physical connection exists, every packet that is received by the Level I module is sent to the Level II protocol module.

Session setup and start up, as performed by Level I, are communication medium dependent. 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 multipoint-to-multipoint mode) the device that has an out-

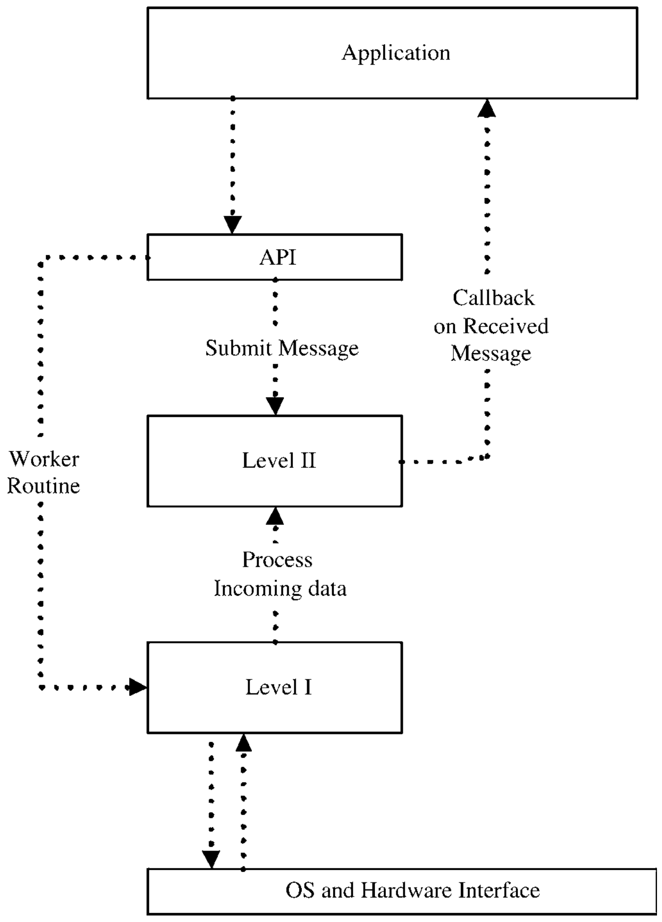

Figure 3. Interaction between components of CentaurusComm protocol (control).

going 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. Every 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 finishes processing 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).

4.1.3. The Level II module

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 acknowledgment mechanism described above. In addition, it provides some rudimentary time synchronization mechanisms along with identification and deletion of old messages. The current version of the Level II code works on PalmOS and Linux (both user space and kernel space).

The Level II module consists of a session based protocol. Session management (setup, shutdown) is still the responsibility of the Level I module. In general, a single session may consist of a single message from each end point to the other. Thus, at most two messages can be transmitted in one session. Under certain conditions, a session may not be able to handle all the data packets sent by an end point. In such cases, the message may span multiple sessions. Multiple sessions may be required if the underlying communication medium does not allow more than 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. Multiple sessions per message may also be required if bad network conditions that cause the loss of the control packet, which contains signaling information. For reasons of time and memory conservation, the CentaurusComm protocol does not provide any mechanism for retransmission of control packets. 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 control information 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. The acknowledgment mechanism ensures that data packets received in the previous session will not be retransmitted.

As described in above, the Level II state machine is triggered by the Level I state machine. Depending on the type of network – IrDA-based or peer-to-peer – the Level II state machine is initialized to the *idle* or the *wait* state, respectively. (We use the abbreviation L2SM to denote the Level II State Machine in the rest of this section.) The L2SM message sequence chart is shown in figure 4. We describe below the message exchanges and corresponding state transitions (as far as the transmission initiation concerned) for both IrDA-based and peer-to-peer networks.

Initial state transitions specific to IrDA-based networks. The L2SM changes to the *wait* state on all slave devices, after receiving a HELO message from the local Level I module. On the designated master device, once the Level I connection has been established with one or more slaves, a HELORESP message is sent to the L2SM. The L2SM on the master device then transitions to the *connected* state. This is done as an additional safeguard to prevent the master device from communicating with devices that do not support the CentaurusComm protocol. As the link latency in IrDA-based networks is quite low (around 500 ms), this safeguard does not cause unacceptable overhead. (However, the overhead in peer-to-peer networks is too high and does not justify use of the safeguard. Hence, the difference in initial state transitions of the L2SM.) In this state, it transmits a POLL message on the IrDA interface to determine if the slave devices have data to send, and transitions back to the *wait* state. When the L2SM on a slave device receives the POLL message, it transitions to the *poll received* state.

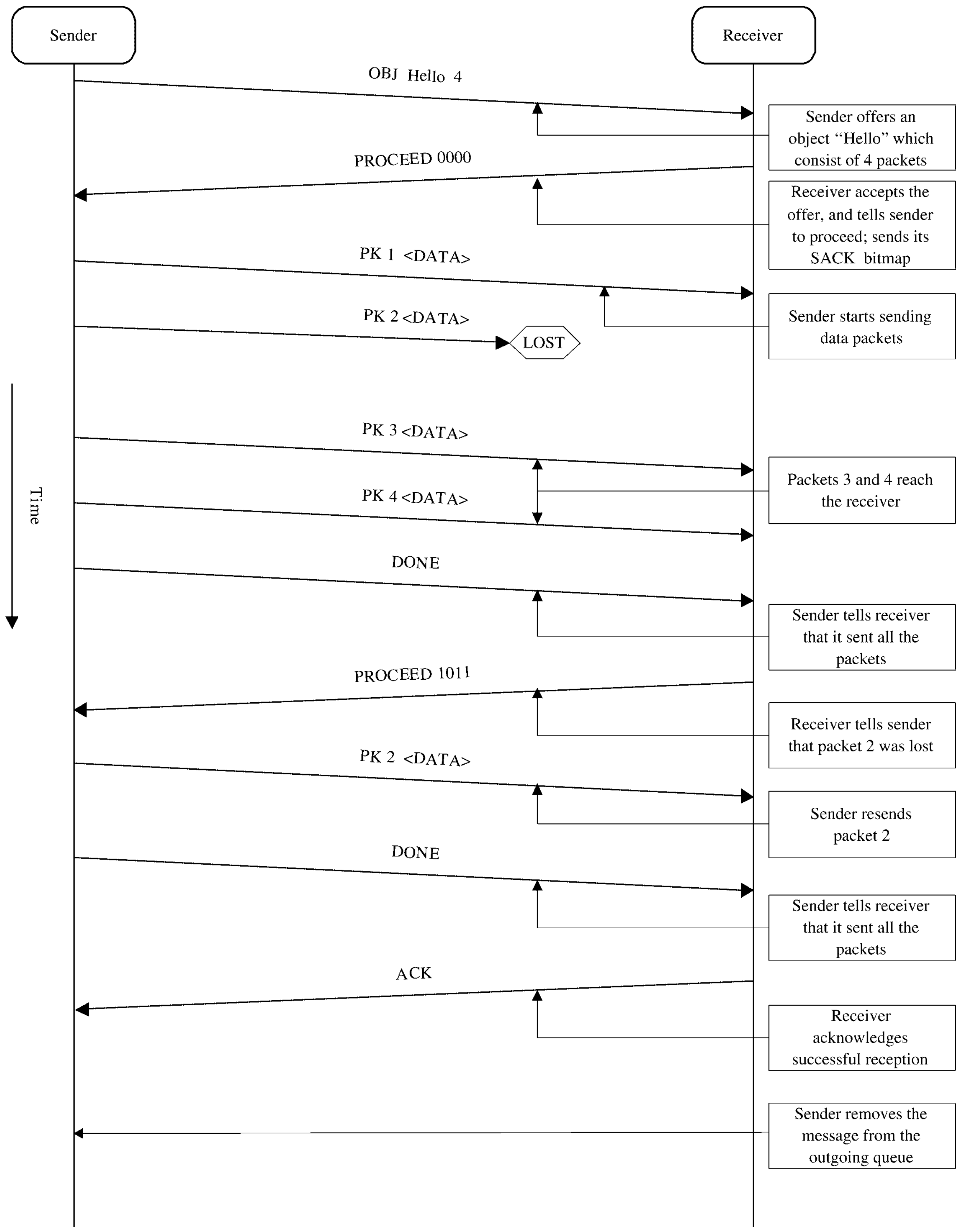
Sender
Receiver
OBJ Hello 4
Sender offers an object "Hello" which consist of 4 packets
PROCEED 0000
Receiver accepts the offer, and tells sender to proceed; sends its SACK bitmap
PK 1 <DATA>
Sender starts sending data packets
PK 2 <DATA>
LOST
Time
PK 3 <DATA>
PK 4 <DATA>
Packets 3 and 4 reach the receiver
DONE
Sender tells receiver that it sent all the packets
PROCEED 1011
Receiver tells sender that packet 2 was lost
PK 2 <DATA>
Sender resends packet 2
DONE
Sender tells receiver that it sent all the packets
ACK
Receiver acknowledges successful reception
Sender removes the message from the outgoing queue

Figure 4. Message exchange sequence for CentaurusComm protocol.

Initial state transition specific to peer-to-peer networks. The L2SM transitions from the *wait* state to the *poll received* state upon receiving a POLL message from the local Level I module.

The remaining state transitions, as described in this and the following subsections are common to both types of networks.

After transitioning to the *poll received* state, the Level II module scans the table of outgoing message objects to find the one that should be delivered to its peer. On small mobile devices, this table typically contains only one slot which is either full or empty, so the selection of the object is trivial. On server class systems, this table will have multiple entries, therefore linear search is used to select the outgoing object. Once the object is selected, the Level II state machine sends an OBJ message (which contains the class of the message object, its size and the timestamp) to its peer (via the Level I module) and transitions to the *send command* state. As soon as the Level I module places the message on the network stack, it informs the Level II module via the SENT message. This forces the Level II state machine back into the *wait* state.

Upon reception of an OBJ message the L2SM on the peer transitions to the *obj received* state. In this state, it examines the class and timestamp on the object and decides whether to accept or reject the object. Objects are rejected if either the class of the object is unacceptable or if the device already has a newer copy of the object of this class. If the device decides to reject this object it sends back a REJ message and transitions back to the *wait* state. Otherwise it sends back a PROCEED message that contains the bitmap, which indicates all the received data packets that are part of this message object. If this is a new message object, this bitmap has all its bits set to 0. In this case, the L2SM transitions to the *proceed sent* state. After the Level I has sent this message to the peer and confirmed the transmission, the L2SM transitions to the *packet receive* state.

If the device that sent the OBJ message receives a REJ message in response, the L2SM transitions to the *ack-rej received* state. In this state, it removes the object from the table of outgoing objects. In addition, it responds with a NOPE message and the L2SM transitions back to the *wait* state. On receiving the NOPE message, the L2SM on the peer transitions to the *nope received* state. In this state, it determines if it has any object to send. If so, it sends it the OBJ message; message exchanges and state transitions following this are exactly as describe in the previous section. If this device has nothing to send to the peer, the session is closed and the L2SM returns to either the *idle* state or *wait* state.

If the device that sent the OBJ message receives a PROCEED message in response, the L2SM transitions to the *proceed received* state. In this state, the Level II module first updates its copy of the bitmap using the one present in the PROCEED message. The L2SM enters the *packet transmit* state. It then starts transmitting those data packets, in the message object, that correspond to the bitmap entries marked 0. The data packets consist of two fields – the packet header and the payload. The packet header consists of two fields – a 2-byte constant corresponding to the string "PK" and a 4-byte slot number. The process of packet transmission is as follows. After determining the correct data packet number to start sending from, the device prepares the packet by setting the destination slot number in the packet header followed by the payload. This packet is copied to the common communication area and a flag is set. This signals to the Level I module that data is ready for transmission. The L2SM transitions to the *wait for packet send* state. After the Level I module has transmitted the packet it informs the Level II module which causes the L2SM to transition back to the *packet transmit* state, after updating the bitmap.

It then checks the bitmap and the object size, and continues to transmit packets as long as the bitmap contains any '0's. We note that the synchronous nature of session set up allows the L2SM to remain in the *packet transmit* state for the duration of transmission instead of switching between this and the *wait for packet send* states. 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.

While L2SM on the source device is in the *packet transmit* state, the destination L2SM is in the *packet receive* state. The header of every data packet received is checked for the "PK" string. The payload is copied into the object receive buffer at the slot indicated in the packet header. The bit corresponding to this slot is set to 1 in the bitmap.

After the source device completes sending all data packets in the message object, L2SM sends a DONE message to the peer and transitions back to the *wait* state. When the peer receives the DONE message, the L2SM transitions to the *done received* state. In this state, it checks its message bitmap; if all the expected data packets from the object have been received then it responds with the ACK message. It also updates the timestamp of the last received object in its table of accepted objects. 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. The L2SM transitions back to the *wait* state. When the L2SM on the data source receives the ACK message, it transitions into the *ack-rej* state. If the data source is the master in IrDA-based networks or the "server" in peer-to-peer networks, it closes the session causing the physical connection to be terminated as well. If the data source is a slave or "client", then it sends a NOPE message to the other side and transitions back to the *wait* state. On receipt of a NOPE message, the master or server L2SM then behaves as described above.

If the destination L2SM receives the DONE message and determines via its bitmap 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 and transitions back to the *proceed sent* state.

4.2. Communication Manager

This component 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. Currently we have communication modules for Bluetooth, IR and CDPD allowing Centaurus to communicate with clients using those communication media.

4.3. 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 is updated dynamically, according to the services registered with the Service Manager, so the client always has the updated list of services. After registering, a client can select a service from the list provided by the Service Manager, 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 leasing. It allows services to register for a certain amount of time. If it does not receive any status update (see section 4.4) within that time, the registration is deleted.

Though the Service Manager does manage some state, most of the state is distributed among the clients and services. In case a Service Manager fails, the services keep pinging it at increasing intervals until it comes up again. Then they reregister themselves with it. Similarly the Service Manager stores a list of services that a client is interested in, so it can be sent status updates from those services. However, every time the client notifies the Service Manager of a service it is interested in, the Service Manager forces the client to update its own CCML description to reflect the new service. As soon as the failed Service Manager is up again or if a replacement Service Manager is used, the clients register themselves. Their CCML description informs the Service Manager what services they are interested in. By forcing the services to be responsible for maintaining their own state, by deregistering any timed out services, and making clients store the list of services they are interested in, the Service Manager manages automatic state recovery.

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 subcomponents.
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.

4.4. 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 printing a document 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, location, 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 renew message to the Service Manager. It accepts requests only from the Service Manager that it is registered with.

4.5. 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 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 its status changes, that is, each time a new service registers, or an existing service deregisters.

By choosing a Service, the client *expresses interest* in it. The Service Manager sends the client the CCML description of the Service. The client displays the CCML file for the user, who can invoke the specified functions on the Service, by choosing one of its interfaces. After the user changes values of certain variables, specified in the CCML for the particular service, the client sends CCML back to the Service Manager in the form of a *command*. The client 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.

```
<!-- 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)*>
```

```
<!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)>
```

Figure 5. The DTD for CCML.

5. Centaurus Capability Markup Language (CCML)

The CCML is divided into *system*, *data*, *addons*, *interfaces*, and *info*, as shown in figure 5.

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. All information regarding the variables and their types are contained in the *data* section. Using the *addons* section, one 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.

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. We have defined two actions for clients, namely, AddService and RemoveService:

- 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 Ser-

vice 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. This section generally causes the variables in the *data* section to change their values.

Other details like the description, and icon for representation are in the *info* section.

6. Implementation

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 implemented one Service Manager and two services for testing. We assume that the client application is installed on the handheld device 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 communication modules and the client, for increased efficiency in resource management. We have found that most of the service discovery architectures are implemented in Java, like Jini [1] and E-Speak [7]. If we decide to incorporate a more sophisticated service discovery technique, integration will be relatively easy as the Service Manager and services are already in Java.

6.1. Communication Manager

The Communication Manager constitutes several modules of CentaurusComm Transport Protocol for communication. 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 4.1. 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.

The functionality of the Communication Manager can be summarized as:

- Read properties file and retrieve poll period and port number to send to Service Manager and port number to receive from Service Manager.
- Based on poll period, periodically poll all clients in the room. If a client replies with a message, forward the message to the Service Manager.
- Wait for message from Service Manager on port. If receive a message on input port, check header and forward to appropriate client and go back to waiting.

6.2. 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.

The control flow for the Service Manager is:

- Read properties file and retrieve port numbers to wait on for Services and Communication Manager.
- Wait for CCML message on both ports.
- If CCML message on port from Communication Manager, then message from a client. Check which client sent the message.
  - If unknown client, then create a new ID and set the action, AddService to ServiceList in the CCML file for the client and returns this CCML to the Communication Manager.
  - If client in already registered, check if *update* or *command*.
    - If it is an *update*, read the list of services that the client is listening to, and pick out the new services that the client was not previously listening to. Sends the CCML of those services to the client via the Communication Manager. Add the Service-Client pair to InterestList.
    - If it is a *command*, extract the name of the service, and send it to the appropriate Service.
- If CCML message on port for services, check the type of the message.
  - If registration message, enter its information into a table and set a timeout. If the service does not send an update without the timeout, delete from the table.
  - If *update* from service, forward this update to all clients in the Service-Client list that are interested in this particular Service.
  - If a *ping* message, renew the lease of the associated Service.
- If the lease period of a service expires and no status update or ping message received from the service, the service is deregister service.

6.3. Client

The client attempts to discover and communicate with the Communication Manager through a certain medium. Once discovered, it registers with the Centaurus system. There are several functions of a client, requesting an action, updating status, and receiving updates. 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.

The main functionality of a client is described as:

- If entering new space, discover a Communication Manager. Send CCML description to Service Manager through Communication Manager. Register with Service Manager.
- If received a CCML message from Communication Manager, check type of message.
  - If it is an *update*, display the status update for the user. If the user makes any changes to the interface of the Service, generate new CCML and send it to the Communication Manager, when next polled.
  - If it is a *command*, validate the CCML. If there is any AddService action, add the value to listening list in CCML description. Set the *update* variable in CCML header. If the action is RemoveService, remove specified Service from list.

6.4. 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 needs only implement a constructor and this commandHandler to be integrated into a Centaurus system.

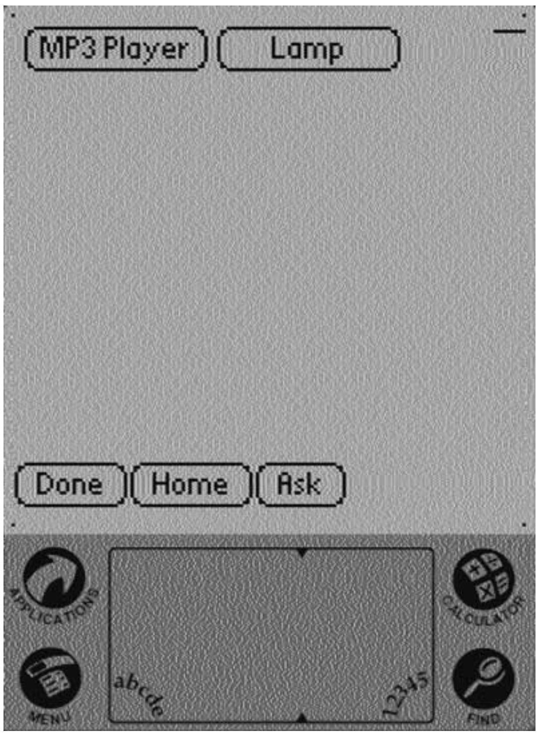

Figure 6. ServiceList.

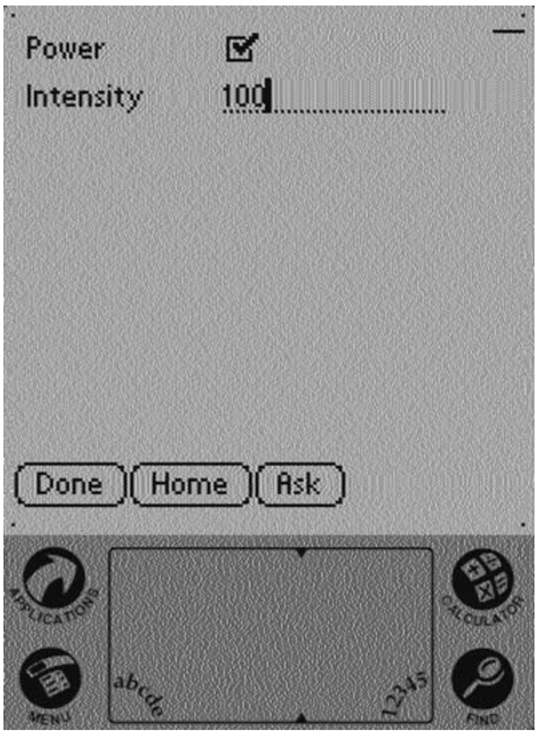

Figure 7. Client turns on Lamp.

The Service class forces a service to register as soon as it starts up. A service 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 list of services. However, 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.

The algorithmic description for the Service class is as follows:

- On starting up, register with Service Manager specified in properties file. Send CCML description, lease period and port number to the Service Manager. Make sure *update* variable is set.
- Wait for message on port.
- On receiving message, check type of message. If it is a *command* from the Service Manager, try to carry out the command. Update CCML to reflect the new status and send it to the Service Manager.
- If no status change within lease period and if need to renew lease, send a lease renew to the Service Manager.

6.4.1. *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 6 shows a screen shot of the ServiceList received by the client (a Palm Pilot).

6.4.2. *Lamp control*

Using X10[2] devices and FireCracker,[3] we were able to control a lamp in the room and figure 11 shows its CCML description. 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 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. A CCML description of a lamp which is currently off is shown in figure 12. Figure 7 shows a screen shot of the Client using the Lamp Service.

Figure 8 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.

[2] `http://www.x10.com`
[3] `http://www.x10.com/welcome/firecracker`

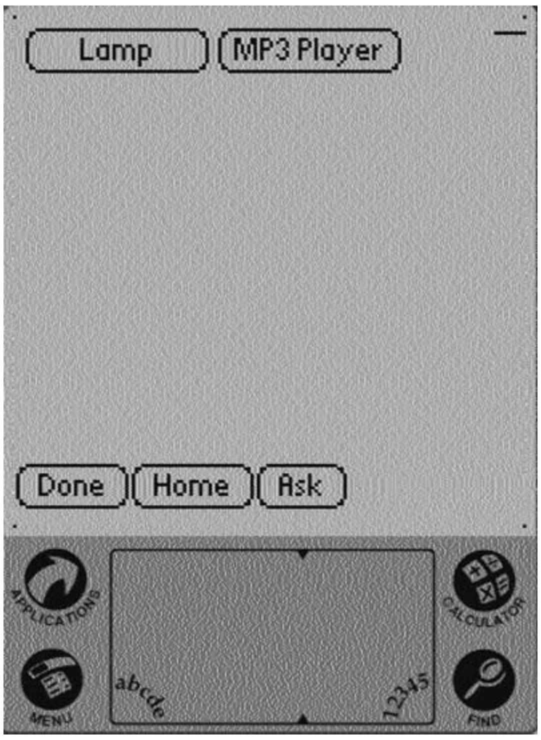

Figure 8. Updated ServiceList.

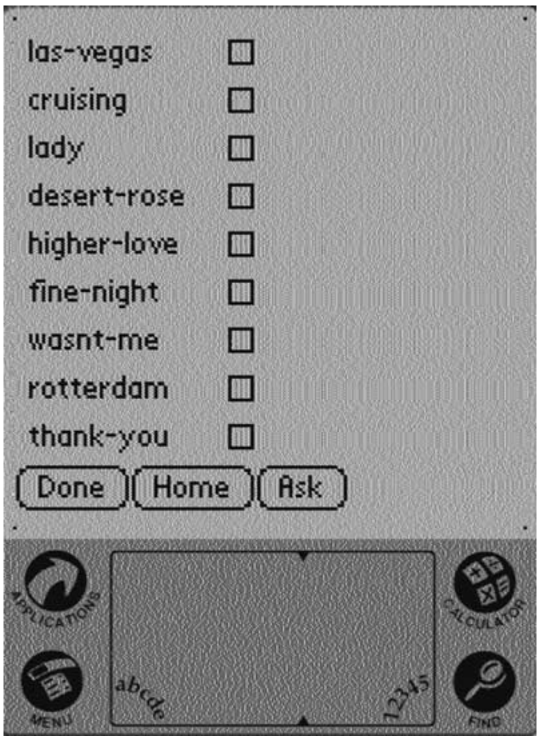

Figure 9. List of songs provided to Client.

6.4.3. MP3 Player

We are using a popular MP3 player for Unix, mpg123,[4] 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 dynamically. It has a number of CCML interfaces, one for each song it can play. The client displays this list for the user and waits for her input. The client returns all songs checked to the service via the Service Manager. Figure 13 shows a CCML command sent to the MP3 Player. The commandHandler function in the MP3 Player checks the CCML interface and reads the songs selected. These songs are checked against the current list of songs. If they are valid, and not currently being played they are fed into *mpg123*. As the status is changed, a status update is sent to the Service Manager.

Figure 9 shows a shot of the list of songs in MP3 format provided by the MP3 Service. In figure 10 we see that the Client has selected 3 songs it wants the MP3 Player to play.

7. Functional interaction between Centaurus components

There are certain crucial functions of the system that require several components to collaborate in order to be completed. This section discusses how a client requests a service, how the ServiceList service is used, and how status updates are propagated through the Centaurus system.

[4] `http://mpg123.org`

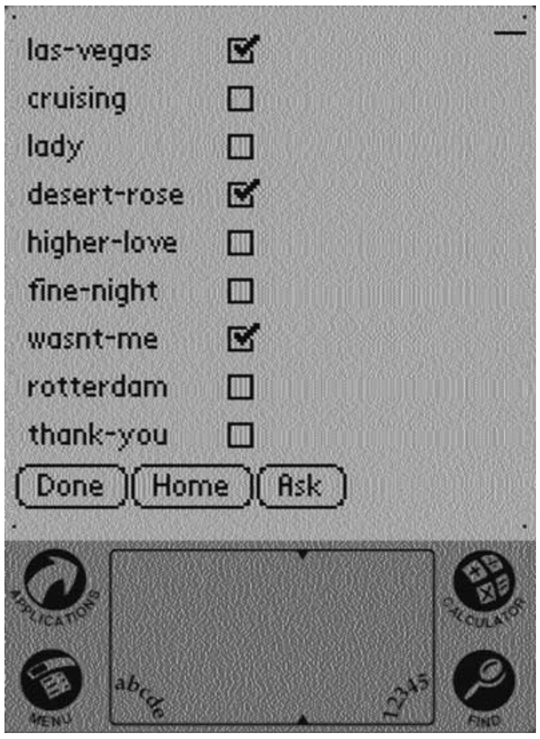

Figure 10. Client selects songs to play.

7.1. Requesting a Service

When a client receives the list of services from the Service Manager, 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,

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE ccml SYSTEM "ccml.dtd">

<!-- Service List  xml file -->
<ccml version="1.0">

<system>
  <full/>
  <command/>
  <interactive/>
  <id name="ServiceList" />
  <manager name="snowy"/>
  <time value="112334" />
  <origin name="Palm-121"/>
  <location name="snowy.cs.umbc.edu"></location>
</system>

<data>
  <attrib  name="MP3Player1.0"   type="bool"
value="true" />
  <attrib  name="Lamp-001"      type="bool"
value="false" />
</data>

<info/>

</ccml>
```

Figure 11. An example ServiceList that shows two services, an MP3 player and a Lamp.

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE ccml SYSTEM "ccml.dtd">

<ccml version="1.0">

<system>
  <full/>
  <update/>
  <interactive/>
  <id name="Lamp-001"  />
  <manager name="snowy"/>
  <time value="112334" />
  <origin name="Lamp-001"/>
  <location name="snowy.cs.umbc.edu"></location>
</system>

<data>
  <attrib  name="Power"      type="bool"    value="false" />
  <attrib  name="Intensity"  type="integer"  value="0" />
</data>

<interfaces>
  <interface name="Powered"/>
</interfaces>

</ccml>
```

Figure 12. A CCML description of the Lamp service.

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.

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE ccml SYSTEM "ccml.dtd">

<ccml version="1.0">

<system>
  <full/>
  <command/>
  <interactive/>
  <id name="MP3Player1.0" />
  <manager name="snowy"/>
  <time value="112334" />
  <origin name="Palm-121"/>
  <location name="snowy.cs.umbc.edu"></location>
</system>

<data>
  <attrib  name="leaving-las-vegas.mp3"   type="bool"
value="true" />
</data>

</ccml>
```

Figure 13. A CCML description of a command sent to the MP3 Player. Each song is an interface which can be turned on or off.

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 propagates the update back to the client.

7.2. Using the ServiceList

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>
<id name = "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>
```

7.3. 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.

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 section 6. Experiments with Bluetooth as the medium were also successful.

9. Ongoing work

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 adhoc network.

9.1. 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 coffeemaker control service during the morning to the user, and in the evening it returns a light control service. 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.

9.2. 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 [19].

9.3. DAML

Most of the information on the web today is described in HTML which is not very powerful or expressive. The World Wide Web Consortium (W3C) developed XML to help describe information better. As a step towards the Semantic Web, DAML 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 because of its expressibility 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.

10. Summary

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.

Acknowledgements

This research was supported in part by the IBM EECOMS program, the DARPA DAML program under contract F30602-97-1-0215, NSF 9875433, NSF 0070802.

References

[1] K. Arnold, A. Wollrath, B. O'Sullivana, R. Scheifler and J. Waldo, *The Jini Specification* (Addison-Wesley, Reading, MA, 1999).

[2] M. Blaze, J. Feigenbaum, J. Ioannidis and A. Keromytis, The role of trust management in distributed systems, in: *Secure Internet Programming*, LNCS Vol. 1603 (Springer, Berlin, 1999) pp. 185–210.

[3] M. Blaze, J. Feigenbaum and J. Lacy, Decentralized trust management, in: *IEEE Proceedings of the 17th Symposium* (1996).

[4] Bluetooth, `http://www.bluetooth.com`

[5] M. Esler, J. Hightower, T. Anderson and G. Borriello, Next century challenges: Data-centric networking for invisible computing, in: *Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)* (ACM Press, August 1999) pp. 256–262.

[6] S.D. Gribble et al., The Ninja architecture for robust Internet-scale systems and services, Computer Networks 35(4) (1999) 473–497.

[7] Hewlett-Packard Company, Internet business solutions: E-Speak, `http://www.e-speak.hp.com`

[8] T. Hodes and R.H. Katz, A document-based framework for Internet application control, in: *Second USENIX Symposium on Internet Technologies and Systems (USITS'99)* (October 1999).

[9] T.D. Hodes, R.H. Katz, E. Servan-Schreiber and L. Rowe, An architecture for a secure service discovery service, in: *3rd ACM/IEEE MobiCom*, Budapest, Hungary (September 1997).

[10] I. Horrocks et al., DAML + OIL language specifications (2001) `http://www.daml.org/2000/12/daml+oil-index`

[11] Infra-red, `http://www.irda.org`

[12] F. Kon, C.K. Hess, M. Roman, R.H. Campbell and M.D. Mickuna, A flexible, interoperable framework for Active spaces, in: *OOPSLA '2000 Workshop on Pervasive Computing*, Minneapolis (October 16, 2000).

[13] D. Norman, The Invisible Computer (MIT Press, 1998).

[14] S.R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan and T. Winograd, ICrafter: A service framework for ubiquitous computing environments, in: *Ubicomp 2001* (2001).

[15] M. Roman and R.H. Campbell, Gaia: Enabling Active Spaces, in: *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark (September 2000).

[16] M. Satyanarayanan, Pervasive computing: Vision and challenges, IEEE Communications (2001).

[17] Stanford Interactive Workspaces Project, `http://graphics.stanford.edu/projects/iwork/`

[18] M. Taylor, W. Waung and M. Banan, Internetwork mobility: The CDPD approach, Prentice Hall professional technical reference, September 1996 (1999).

[19] J. Undercoffer, A. Cedilnik, F. Perich, L. Kagal and A. Joshi, A secure infrastructure for service discovery and management in pervasive computing, Mobile Networks and Applications (2002).

[20] M. Weiser, The computer for the twenty-first century, Scientific American (September 1991) 94–100.

Lalana Kagal is a Ph.D. student and a Graduate Research Assistant in the Department of Computer Science and Electrical Engineering at the University of Maryland Baltimore County. She obtained both a Bachelor of Science degree and a Master of Science degree in computer science from the Pune University (India) in 1994 and 1996, respectively. Her research interests are in the fields of artificial intelligence, security and mobile computing. She is a student member of ACM and AAAI.
E-mail: lkagal1@cs.umbc.edu

Vlad Korolev is a graduate student in the Department of Computer Science and Electrical Engineering at the University of Maryland Baltimore County. He has got a Bachelor of Science degree in computer science from the University of Maryland Baltimore County. His research interests are in the fields of mobile computing and computer graphics.
E-mail: vkorol1@cs.umbc.edu

Sasikanth Avancha is a Ph.D. student of computer science in the Department of Computer Science and Electrical Engineering at the University of Maryland Baltimore County. His research interests include wireless networks, mobile computing, and distributed systems. He obtained his M.S. degree in computer science from UMBC and his Bachelor of Engineering degree in computer science and engineering from Bangalore University, India. He has over 5 years of experience in network software development and maintenance.
E-mail: savanc1@cs.umbc.edu

Anupam Joshi is an Associate Professor of Computer Science and Electrical Engineering at UMBC. Earlier, he was an Assistant Professor in the CECS department at the University of Missouri, Columbia. He obtained a B.Tech. degree in electrical engineering from IIT Delhi in 1989, and a Masters and Ph.D. in computer science from Purdue University in 1991 and 1993, respectively. His research interests are in the broad area of networked computing and intelligent systems. His primary focus has been on data management for mobile systems in general, and most recently on data management in mobile ad hoc networks. He has created intelligent agent based middleware to support mobile access to networked computing and multimedia information resources. He is also interested in Data/Web Mining and Semantic Web, where he has worked on applying soft computing techniques to personalize the web space. His other interests include content-based retrieval of video data from networked repositories, and networked HPCC. He has published over 50 technical papers, and has obtained research support from NSF, NASA, DARPA, DoD, IBM, AetherSystens, HP, AT&T and Intel. He has presented tutorials in conferences, served as a guest editor for special issues for IEEE Personal Communications, Communications of the ACM etc., and serves as an Associate Editor of IEEE Transactions on Fuzzy Systems. At UMBC, Joshi teaches courses in operating systems, mobile computing, and Web mining. He is a member of IEEE, IEEE-CS, and ACM.
E-mail: joshi@cs.umbc.edu

Tim Finin is a Professor in the Department of Computer Science and Electrical Engineering and the director of the Institute for Global Electronic Commerce at the University of Maryland Baltimore County (UMBC). He has over 30 years of experience in the applications of AI to problems in information systems, intelligent interfaces and robotics. He holds degrees from MIT and the University of Illinois. Prior to joining the UMBC, he held positions at Unisys, the University of Pennsylvania, and the MIT AI Laboratory. Finin is the author of over 140 refereed publications and has received research grants and contracts from a variety of sources. He has been the past program chair or general chair of several major conferences, is a former AAAI councilor and is AAAI's representative on the board of directors of the Computing Research Association.
E-mail: finin@cs.umbc.edu

Yelena Yesha received the B.Sc. degree in computer science from York University, Toronto, Canada, in 1984, and the M.Sc. and Ph.D. degrees in computer and information science from The Ohio State University in 1986 and 1989, respectively. Since 1989 she has been with the Department of Computer Science and Electrical Engineering at the University of Maryland Baltimore County, where she is presently a Verizon Professor. In addition, from December 1994 through August 1999 Dr. Yesha served as the Director of the Center of Excellence in Space Data and Information Sciences at NASA. Her research interests are in the areas of distributed databases, distributed systems, mobile computing, digital libraries, electronic commerce, and trusted information systems. She published 8 books and over 100 refereed articles in these areas. Dr. Yesha was a program chair and general co-chair of the ACM International Conference on Information and Knowledge Management and a member of the program committees of many prestigious conferences. She is a member of the editorial board of the Very Large Databases Journal, and the IEEE Transaction on Knowledge and Data Engineering, and is the editor-in-chief of the International Journal of Digital Libraries. During 1994, Dr. Yesha was the Director of the Center for Applied Information Technology at the National Institute of Standards and Technology. Dr. Yesha is a senior member of IEEE, and a member of the ACM.
E-mail: yeyesha@cs.umbc.edu