Rover Technology: Enabling Scalable Location-Aware Computing

Suman Banerjee, Sulabh Agarwal, Kevin Kamel, Andrzej Kochut, Christopher Kommareddy,
Tamer Nadeem, Pankaj Thakkar, Bao Trinh, Adel Youssef, Moustafa Youssef,
Ron Larsen, A. Udaya Shankar, Ashok Agrawala

MIND Lab, UMIACS and Department of Computer Science
University of Maryland, College Park, MD 20742, USA

UMIACS-TR 2001-89 and CS-TR 4312
December 2001

Abstract

Location-aware computing involves the automatic tailoring of information and services based on the current location of the user. We have designed and implemented Rover, a system that enables location-based services, as well as the traditional time-aware, user-aware and device-aware services. To achieve system scalability to very large client sets, Rover servers are implemented in an "action-based" concurrent software architecture that enables fine-grained application-specific scheduling of tasks. We have demonstrated feasability through implementations for both outdoor and indoor environments on multiple platforms.

1 Introduction

A user is shopping in a mall. On entering a store, he pulls out a PDA and browses through detailed information about the products on display. Satisfied with the information, through the PDA, he makes an online purchase of the items of interest, that will be subsequently shipped to his home directly. As he walks on to the next store, which happens to be a video rental store, information on newly-released movies in his favorite categories are downloaded automatically into his PDA, along with their availability information. He chooses a couple of these movies and indicates that he will pick them up at the storefront. His membership discounts are applied to the bill, and he confirms the charge to his credit card.

The intriguing aspect of this scenario is the automatic tailoring of information and services based on the current location of the user. We refer to this paradigm as **location-aware computing**. The different technology components needed to realize location-aware computing are present today, powered by the increasing capabilities of mobile personal computing devices and the increasing deployment of wireless connectivity (IEEE 802.11 wireless LANs [7], Bluetooth [1], Infra-red [2], Cellular services, etc.)

What has hindered its ubiquitous deployment is the lack of system-wide integration of these components in a manner that scales with large user populations. In this paper, we describe the design and initial implementation experience of such a system, which we call *Rover*, and discuss the impact such a system can have on the next generation of applications, devices, and users.

Rover enables services with the following characteristics:

- Location-aware, in addition to the more traditional notions of time-aware, user-aware, and device-aware. Rover has a location service that can track the location of every user, either by automated location determination technology (for example, using signal strength or time difference) or by the user manually entering current location (for example, by clicking on a map).

- Available via a variety of wireless access technologies (IEEE 802.11 wireless LANs, Bluetooth, Infra-red, cellular services, etc.) and devices (laptop, PDA, cellular phone, etc.), and allows roaming between the different wireless and device types. Rover dynamically chooses between different wireless links and tailors application-level information based on the device and link layer technology.

- Scales to a very large client population, for example, thousands of users. Rover achieves this through fine-resolution application-specific scheduling of resources at the servers and the network.

We will use a **museum tour application** as an example to illustrate different aspects of Rover. We consider a group of users touring the museums in Washington D.C. At a Rover registration point in a museum, each user is issued a handheld device with audio and video capabilities, say an off-the-shelf PDA available in the market today. Alternatively, if a user possesses a personal device, he can register this device and thus gain access to Rover. The devices are trackable by the Rover system. So as a user moves through the museum, information on relevant artifacts on display are made available to the user's device in various convenient forms, for example, audio or video clips streamed to the device. Users can query the devices for building maps and optimal routes to objects of their interest. They can also reserve and purchase tickets for exhibitions and shows in the museum later in the day. The group leader can coordinate group activities by sending relevant group messages to the users. Once deployed, the system can be easily expanded to include many other different services to the users.

The next section gives a description of the kinds of services that are available through Rover. The successive sections provide an overview of the Rover architecture and a description of a concurrent software architecture that has been used for system scalability. The following sections expand on particular aspects of Rover, including clients, servers, data management and multi-Rover systems. Then we describe our initial implementation experience and conclude with ongoing and future work.

2 Rover Services

The services provided by Rover to its users can be classified as follows:

- *Basic data services:* Rover enables a basic set of data services in different media formats, including text, graphics, audio, and video. Users can subscribe to specific data components dynamically through the device user interface. Depending on the capabilities of the user's device, only a select subset of media formats may be available to the user. This data service primarily involves one-way interaction; depending on user subscriptions, appropriate data is served by the Rover system to the client devices.

- *Transactional services:* These services have commit semantics that require coordination of state between the clients and the Rover servers. A typical example is e-commerce interactions.

Services that require *location manipulation* are a particularly important class of data services in Rover. Location is an important attribute of all objects in Rover. The technique used to estimate the location of an object (some techniques are described in the Appendix) significantly affects the granularity and accuracy of the location information. Therefore an object's location is identified by a tuple of *Value*, *Error*, and *Timestamp*. The error identifies the uncertainty in the measurement (value). The timestamp identifies when the measurement was completed. The accuracy of the location information is relevant to the *context* of its use. For example, an accuracy of 4 meters is adequate to provide walking directions from the user's current location to another location about 500 meters away. However, this same accuracy is inadequate to identify the exhibit in front of the user. User input in these cases, helps significantly improve the accuracy of user location information.

Map-based services are an important component of location manipulation services. Rover maps can be subject to various operations before being displayed to users:

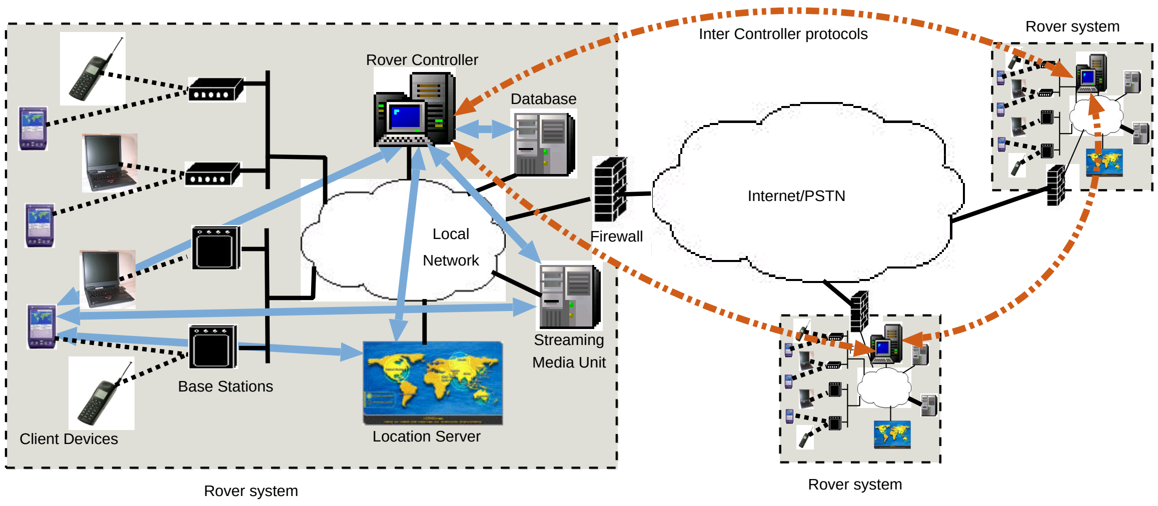

Figure 1: Physical architecture of the Rover System

- *Filter:* Objects in a Rover map have a set of attributes that identify certain properties of the objects. Depending on the user's context (which indicates the user's interests), filters are generated for the attribute values of interest to the user. These filters are applied to maps to select the appropriate subset of objects to display to the user. For example, one user may be interested in only the restaurants in a specific area, while another user needs to view only the museum and exhibition locations. The filters can be dynamically changed to appropriately change the objects being displayed on the map.

- *Zoom:* The zoom level of a displayed map identifies its granularity. The zoom level at a client device is chosen based on the user's context. For example, a user inside a museum gets a detailed museum map, but when the user steps outside the museum, he gets an area map of all the museums and other points of interest in the geographic vicinity. The zoom level can be implemented as an attribute of objects, and appropriate filters can then be applied to display a map at the desired zoom level.

- *Translate:* This functionality enables the map service to automatically update the view of the displayed map on the client device as the user moves through the system. When the location of the user moves out of the central region of the currently displayed map, the system prepares a new map display, that is appropriately translated from the previously displayed map.

3 Rover Architecture

A Rover system, depicted in Figure 1, consists of the following entities:

- **End-users** of the system. Rover maintains a **user profile** for each end-user, that defines specific interests of the user and is used to customize the content served.

- **Rover-clients** are the client devices through which users interact with Rover. They are typically small wireless handheld units with great diversity of capabilities in regard to processing, memory and storage, graphics and display, and network interface. Rover maintains a *device profile* for each device, identifying its capabilities and thus, the functionality available at that device.

- **Wireless access infrastructure** provides wireless connectivity to the Rover clients. Possible wireless access technologies include IEEE 802.11 based wireless LANs, Bluetooth and Cellular services. For certain QoS guarantees, additional mechanisms need to be implemented at the access points of these technologies for controlled access to the wireless interface.

- **Servers** implement and manage the various services provided to the end-users. The servers consist of the following:

  The server system consists of the following set of devices:

  - **Rover controller**: is the “brain” of the Rover system. It provides and manages the different services requested by the Rover clients. It schedules and filters the content sent to the clients based on user and device profiles and their current locations.
  - **Location server**: is a dedicated unit responsible for managing the client device location services within the Rover system. Alternatively, an externally available location service (e.g. as proposed in [3]) can also be used.
  - **Media streaming unit**: provides the streaming of audio and video content to the clients. In fact, it is possible to use many of the off-the-shelf streaming-media units that are available today and integrate them in the Rover system.
  - **Rover database**: stores all content delivered to the Rover clients. It also serves as the stable store for the state of the users and clients that is maintained by the Rover controller.
  - **Logger**: interacts with all the Rover server devices and receives log messages from their instrumentation modules.

There are two potential bottlenecks that can hinder the scalability of such a system to large user populations. One is the server system because it needs to handle a very large number of client requests with tight real-time constraints. Another potential bottleneck is the bandwidth and latency of the wireless access points.

For a server to handle such a large volume of real-time requests, in addition to adequate compute power and appropriate data structures, it must have **fine-grained real-time application-specific scheduling** of tasks to efficiently manage the available resources, both processing and bandwidth. This leads us to divide server devices into two classes — **primary servers**, which directly communicate with the clients, and **secondary servers**, which do not directly communicate with clients but interact with primary servers to provide back-end capabilities to the system. The Rover controller, location server and media streaming unit are examples of primary servers, while the Rover database and the logger are examples of secondary servers.

In order to meet the performance objectives, only the primary servers need to implement the fine-grained real-time task scheduling mechanism. We have defined a concurrent software architecture called the *Action model* that provides such a scheduling mechanism, and implemented the Rover controller accordingly. The Action model, explained below, avoids the overheads of thread context switches and allows a more efficient scheduling of execution tasks.

The Rover system exports a set of well defined interfaces through which it interacts with the heterogeneous world of users and devices with their widely varying requirements and capabilities. Thus new and different client applications can be developed by third-party developers to interact with the Rover system.

A Rover system represents a single domain of administrative control that is managed and moderated by its Rover controller. A large domain can be partitioned into multiple administrative domains each with its own Rover system, much like the existing Domain Name System [9]. For this multi-Rover system, we define protocols that allow interaction between the domains. This enables users registered in one domain to roam into other domains and still receive services from the system.

4 Action model

In order to achieve fine-grained real-time application-specific scheduling, the Rover controller is built according to a concurrent software architecture we call the **action model**. In this model, scheduling is done in "atomic" units called actions. An **action** is a "small" piece of code that does not have any intervening I/O operations. Once an action begins execution, it cannot be pre-empted by another action. Consequently, given a specific server platform, it is easy to accurately bound the execution time of an action. The actions are executed in a controlled manner by an Action Controller.

We use the term **server operation** to refer to a transaction, either client- or administrator-initiated, that interacts with the Rover controller; examples in the museum scenario would be *registerDevice*, *getRoute* and *locateUser*. A server operation consists of a sequence (or more precisely, a partial order) of actions interleaved by asynchronous I/O events. Each server operation has exactly one "response handling" action for handling all I/O event responses for the operation; i.e., the action is eligible to execute whenever an I/O response is received.

A server operation at any given time has zero or more actions eligible to be executed. A server operation is in one of the following three states:

- Ready-to-run: At least one action of the server operation is eligible to be executed but no action of the server operation is executing.
- Running: One action of the server operation is executing (in a multi-processor setup, several actions of the operation can be executing simultaneously).
- Blocked: The server operation is waiting for some asynchronous I/O response and no actions are eligible to be executed.

The Action Controller uses administrator-defined policies to decide the order of execution of the set of eligible actions. The scheduling policy can be a simple static one, such as priorities assigned to server operations, but it can equally well be time based, such as earliest-deadline-first or involving real-time cost functions. In any case, the controller picks an eligible action and executes it to completion, and then repeats, waiting only if there are no eligible actions (presumably all server operations are waiting for I/O completions).

The management and execution of actions are done through a simple **Action API** defined as follows:

- *init (action_id, function_ptr)*: This routine is called to initialize a new action (identified by *action_id*) for a server operation. *Function_ptr* identifies the function (or piece of code) to be executed when the action runs.
- *run (action_id, function_parameters, deadline, deadline_failed_handler_ptr)*: This routine is called to mark the action as eligible to run. *Function_parameters* are the parameters used in executing this instance of the action. *Deadline* is optional and indicates the time (relative to the current time) by which the action should be executed. This is a soft deadline, that is, its violation leads to some penalty but not system failure. If the action controller is unable to execute the action within the deadline, it will execute the function indicated by *deadline_failed_handler_ptr*. This parameter can be *NULL*, indicating that no compensatory steps are needed.
- *cancel (action_id, cancel_handler_ptr)*: This routine is called to cancel a ready-to-run action provided it is not executing. *Cancel_handler_ptr* indicates a cleanup function. It can be *NULL.*

Actions vs Threads: Our need to scale to very large client populations made us adopt the action model rather than the more traditional thread model. We now provide some experimental justification.

There are several ways to use a thread model to implement the Rover controller. One is to implement each server operation as a separate thread. Another is to have a separate thread for each user. Both of these imply a large number of simultaneously active threads as we scale to large user populations, resulting in large overheads for thread switching. A more sensible approach is to create a small set of "operator" threads that execute all operations, for example, one thread for all *registerDevice* operations, one for all *locateUser* operations, and so on. Here the thread switching overhead is modest but there are drawbacks. One is that, depending on the threads package, it restricts our ability to optimize thread scheduling, especially as we transit to time-based (rather than priority based) scheduling. More importantly, because each operator thread executes its set of operations in sequence, this approach severely limits our ability to optimally schedule the eligible actions within an operation and across operations. Of course, each thread could keep track of all its eligible actions and do scheduling at the action level, but this is essentially recreating the action model within each thread.

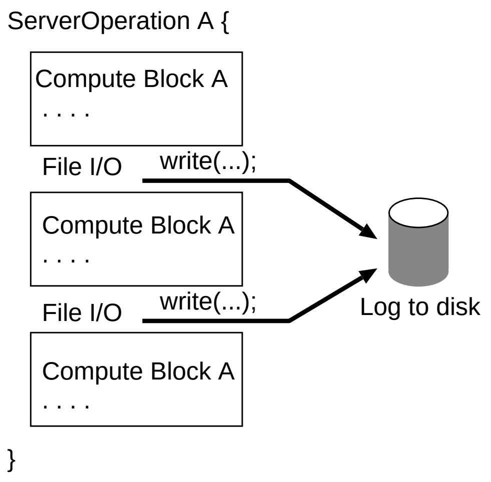

Figure 2: Scenario A has 10,000 processor-bound server operations where computation is interleaved with file write operations

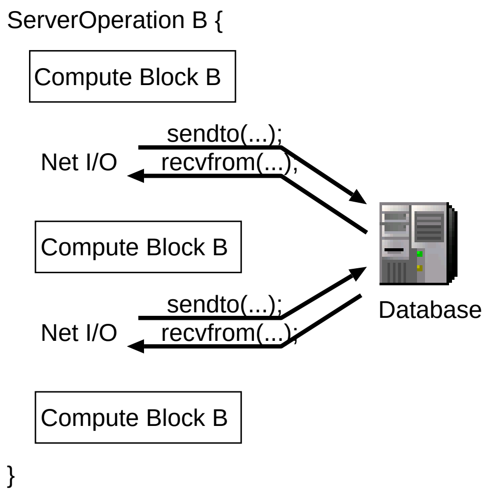

Figure 3: Scenario B has 100 I/O bound server operations where computation is interleaved with network I/O interactions

We compare the performance of action-based and thread-based systems through implementation. In this paper, we consider two kinds of server operations, one processor-bound and the other, I/O-bound, both of which appear in the context of the Rover controller. Using these server operations, we construct two corresponding scenarios:

1. *Scenario A:* This is a computation-intensive scenario and has 10,000 processor-bound server operations, where each of the server operations has three compute blocks, interleaved with two file write operations (see Figure 2). In each of these server operations, the second and the third I/O compute block do not need to await for the prior file I/O write operation to complete.

2. *Scenario B:* This is an I/O-intensive scenario and has 100 I/O-bound server operations, where each of the server operations has three compute blocks, interleaved with two network I/O operations (see Figure 3). In each of these server operations, the second and third compute blocks can be initiated only after the completion of the prior network I/O operation. The network I/O interaction was implemented using UDP. Since our focus is on the comparison of the action-based versus the thread-based systems, we avoided issues of packet loss and re-transmissions by only considering those experiments where no UDP packets were lost in the network.

We consider two execution platforms, referred to as M1 and M2 in the paper. M1 comprises of a Intel Pentium III (600 MHz) processor and 96 MB of RAM which runs Linux. M2 comprises of a Sun Ultra 5, with a Sparc (333 MHz) processor and 128 MB RAM and runs Solaris. For the thread-based implementation, we used the LinuxThreads library for the M1 platform and the Pthreads library for the M2 platform, both of which are implementations of the Posix 1003.1c threads package. This total execution time for the three compute blocks in each server operation A was 0.1518 ms for M1 and 0.9069 ms for M2. The ping network latency for the network I/O in server operation B varied between 30-35 ms.

We compared performances of an action-based implementation and a thread-based implementation of the two scenarios for the different platforms. In the action-based implementation each compute block is implemented as a separate action. In the thread-based implementation, we experimented with a different number of threads, where each thread executed an equal number of server operations for perfect load balancing between the different threads.

		Action-based	Thread-based (threads used)				
Scenario	Machine Specifications		1	5	10	50	100
A (Fig. 2)	M1: Pentium/Linux	**24.27**	299.36	299.93	300.46	304.50	310.31
A (Fig. 2)	M2: Sparc/Solaris	**62.82**	1000.9	1012.54	1041.60	1012.83	1031.25
B (Fig. 3)	M1 Controller, M2 Database	**11.61**	3711.94	1302.20	1011.49	893.10	728.30

Table 1: Comparisons of overheads for action-based and thread-based systems (in ms)

In Table 1, we present the overheads obtained in each case, where the overhead is the total execution time minus the fixed, identical and unavoidable computation/communication costs for the two scenarios. We report the mean execution overheads of a large number of runs, which were required to obtain low variance. For the computation-intensive server operations there are very little performance gains in trying to overlap computation with communication (file I/O), and is not substantial enough to justify the overheads of a multi-threaded implementation. Therefore, a thread-based system with a single thread achieves the best performance among the thread-based implementations.

For the I/O intensive server operations, using a multi-threaded implementation is useful, since computation and communication can be overlapped. Consequently, the best performance for the thread-based system is achieved, when the maximum number of threads are used (one thread for each server operation).

As can be observed in both scenarios, the action-based implementation still achieves significantly (about an order of magnitude) less overheads as compared to the best thread-based implementation.

5 Rover Clients

The client devices in Rover are handheld units of varying form factors, ranging from powerful laptops to simple cellular phones. They are categorized by the Rover controller based on attributes identified in the device profiles, such as display properties — screen size and color capabilities, text and graphics capabilities, processing capabilities — ability to handle vector representations and image compression, audio and video delivery capabilities and user interfaces. The Rover controller uses these attributes to provide responses to clients in the most compatible formats.

For the wireless interface of client devices, we have currently considered two link layer technologies — IEEE 802.11 Wireless LAN and Bluetooth. Bluetooth is power efficient and is therefore better at conserving client battery power. According to current standards, it can provide bandwidths of upto 2 Mbps. In contrast, IEEE 802.11 wireless is less power-efficient but is widely deployed and can currently provide bandwidths of upto 11 Mbps.

In areas where these high bandwidth alternatives are not available, Rover client devices will use the lower bandwidth air interfaces provided by cellular wireless technologies that use CDMA [11] or TDMA based techniques. In particular, cellular phones can connect as clients to Rover, which implies that the Rover system interfaces with cellular service providers.

Different air-interfaces may be present in a single Rover system or in different domains of a multi-Rover system. In either case, *software radios* [8] is an obvious choice to integrate different air-interface technologies.

While the location management system is not tied to a particular air interface, certain properties of specific air interfaces can be leveraged to better provide location management (discussed in the Appendix).

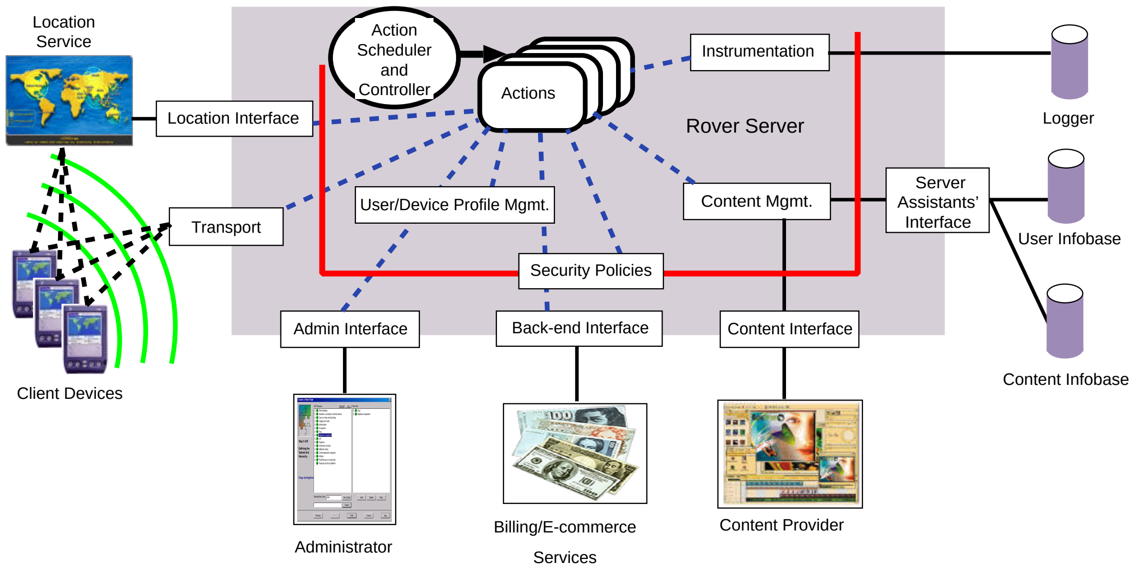

Figure 4: Logical architecture of a Rover system

6 Rover Controller

The interaction of the Rover controller with all other components of the system is presented in Figure 4. The Rover controller interacts with the external world through the following interfaces:

- *Location Interface:* This interface is used by the Rover controller to query the location service about the positions of client devices. The location of a device is defined as a tuple representing the estimate of its position (either absolute or relative to some well-known locations), the accuracy of the estimate, and the time of location measurement. Depending on the technology being used to gain position estimates, The accuracy of the estimate depends on the particulars of the location technology, for example, GPS [6], IEEE 802.11 signal strength, signal propagation delays, etc. Rover takes into account this accuracy information when making location-based decisions.

- *Admin Interface:* This interface is used by system administrators to oversee the Rover system, including monitoring the Rover controller, querying client devices, updating security policies, issuing system specific commands, and so on.

- *Content Interface:* This interface is used by the content provider to update the content that is served by the Rover controller to the client devices. Having a separate content interface decouples the data from the control path.

- *Back-end Interface:* This interface is used for interaction between the Rover controller and certain external services as may be required. One such service is e-commerce, by which credit card authorization for various purchases can be made. These services would typically be provided by third-party vendors.

- *Server Assistants Interface:* This interface is used for interaction of the Rover controller with the secondary servers. e.g. the database and the streaming media unit.

- *Transport Interface:* This is the communication interface between the Rover controller and the clients, which identify data formats and interaction protocols between them.

7 Rover Database

The database in Rover consists of two components, which together decouple client-level information from the content that is served.

One component of the database is the **user infobase**, which maintains user and device information of all active users and devices in the system. It contains all client-specific contexts of the users and devices, namely profiles and preferences, client location, and triggers set by the clients. This information changes at a fairly regular rate due to client activities, e.g. the client location alters with movements. The Rover controller has the most updated copy of this information and periodically commits this information to the database. For many of these data items (e.g. client location), the Rover controller lazily updates the database. These are termed as *volatile data* since any change to these data items are not guaranteed to be accurately reflected by the system across system crashes. For some others, (e.g. new client registration) the Rover controller commits this information to the database before completing the operation. These are termed as *non-volatile data*. The Rover controller, identifies some parts of the data to be volatile, so as to avoid very frequent database transactions. The Rover controller does not guarantee perfect accuracy of the volatile data, and thus trades off accuracy with efficiency for these data components.

The other component in the database is the **content infobase**. This stores the content that is served by the Rover controller and changes less frequently. The content provider of the Rover system is responsible for keeping this infobase updated. In the museum example, this component stores all text and graphical information about the various artifacts on display.

The Rover database implements an extended-SQL interface that is accessed by the Rover controller. Apart from the usual SQL functionality, it also provides an API for retrieval of spatial information of different objects and clients in the system.

The transactions of the Rover controller with the database are executed on behalf of the different server operations. The transactions, by definition, are executed atomically by the database. Additionally, each transaction is identified by two different flags that identify certain properties for execution, as follows:

- Lock-Acquiring: If this flag is set, the transaction is required to acquire relevant locks, on behalf of the server operation, to read or write data to the database. It also requires that these locks will be released by the server operation prior to its termination at the Rover controller.

- Blocking: If a transaction issued by a server operation is unable to access or modify some data due to locks being held by other server operations, it can either block till it successfully reads the data, or it returns immediately to the server operation without successfully execution. If the Blocking flag is set for a transaction, then the first option is chosen for the transaction.

To avoid deadlocks, server operations acquire the relevant locks on data items stored in the database using a *Two Phase Locking* protocol with a lexicographic ordering of lock acquiring for data items. It is important to note that server operations may need to acquire locks at the database, if and only if they need to access the stored data through multiple transactions and all these transactions need to have the same data view. This is not required for the vast majority of server operations that either make a single database transaction, or do not need its multiple transactions to have identical views. None of the server operations in the current implementation of Rover, required to acquire locks at the database. The transactions themselves might acquire and release locks at the database during their execution, which are not visible to the server operations at the Rover controller.

8 Multi-Rover System

A single Rover system comprises of a single Rover controller, other server devices (e.g., Rover database and Rover streaming media unit), and a set of Rover clients. A single system is sufficient for management of Rover-clients in a zone of single administrative control. For example, consider a Rover system in a single museum. All artifacts and objects on display in the museum are managed by a single administrative entity. There is a single content provider for this system and a single Rover system is appropriate to serve all visitors to this museum.

However, each separate museum has its independent administrative authority. Therefore, we can have a separate Rover system for each of the different museums that are administered separately by each museum authority. This allows a decentralized administration of the independent Rover systems, locally by each museum authority. However, it is important to provide a seamless experience to visitors as they roam from museum to museum. A multi-Rover system is a collection of independent Rover systems that peer with each other to provide this seamless connectivity to the user population.

The design of a multi-Rover system is similar in spirit to the Mobile IP [10] solution to provide network layer mobility to devices. Each client device has a *home* Rover system to which it is registered. As the device physically moves into the zone of a different, or *foreign* Rover system, it needs to authenticate itself with the Rover controller of the foreign system. Based on administrative policies, the two Rover systems have service-level agreements that define the services that they will provide to clients of each other.

When the Rover controller of a system detects a foreign client device, it first checks whether it has an appropriate service-level agreement with the Rover controller of the device's home system. If one exists, the Rover controller of the foreign system requests transfer of relevant state about the client device from the Rover controller of the home system and subsequently provides necessary services to it. Rover controllers of different Rover system use the Inter-Controller protocols to interact.

9 Initial Implementation

We have successfully built Rover prototype systems and tested them in the campus of University of Maryland College Park. The implementation has been demonstrated for both indoor and outdoor environments. A preliminary test implementation was developed on Windows based systems (Windows 2000 for the controller and Windows CE for the client devices). The current implementation of the Rover system has been developed under the Linux operating system. The Rover controller is implemented on a Intel Pentium machine running RedHat Linux 7.1 and the clients are implemented on Compaq iPAQ Pocket PC (model H3650) running the Familiar distribution (release versions 0.4 and 0.5) of Linux for PDAs[1]. Wireless access is provided using IEEE 802.11 wireless LANs. Each Compaq iPAQ is equipped with a wireless card which is attached to the device through an expansion sleeve.

We have experimented with a set of 8 client devices and have tested various functionalities of the system.

[1]Available at http://familiar.handhelds.org

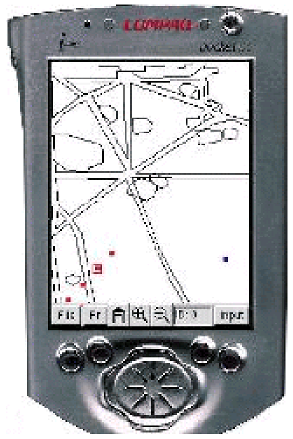

Figure 5: View of the display of a Rover-client

For our outdoor experiments, we interfaced a GPS-device (Garmin e-Trex) to the Compaq iPAQs and obtained device location accuracy of between 3-4 meters. The display of the iPAQ Rover-client displays the locations of the different users (represented by the dots) on the area map as shown in Figure 5.

The indoor Rover system is implemented for the 4th floor of the A.V. Williams Building (where the Computer Science Department is located), whose map is shown in Figure 6. In this implementation, the location service is being provided using signal strength measurements from different base stations. There are about 12 base stations that are distributed all over the building and typically the client device can receive beacons from five or six of the base stations. We are able to get an accuracy of better than a meter in this environment, using very simple signal-strength based estimation techniques.

In both these cases, we implemented the basic functionality of the Rover system. They include:

- User activation/de-activation and device registration/de-registration procedures.
- Periodic broadcast of events of interest from the Rover controller to the users in specific locations.
- Interaction between users. This can be either simple text messaging or voice chat. Users can optionally make their location visible to other users. In the museum example, a tour group coordinator can use this feature to locate all the other members of the group.
- Users can request alerts from the Rover controller when certain conditions are met. The conditions may be time, location or context dependent. This can be used to provide notification to ticket holders of an approaching show time. Clearly, for the users who are further away from the show venue, this notification needs to be provided early enough, so that they have enough time to reach the venue.
- An administrator's console allows a global view of all users and their locations in the system. The administrator can directly interact with all or a specific subset of the users based on the location or other attributes of the users.

Currently, we are implementing more functionality in the Rover controller.

10 Conclusions and Future Work

Rover is currently available as a deployable system using specific technologies, both indoors and outdoors. Our final goal is to provide a completely integrated system that operates under different technologies, and allows

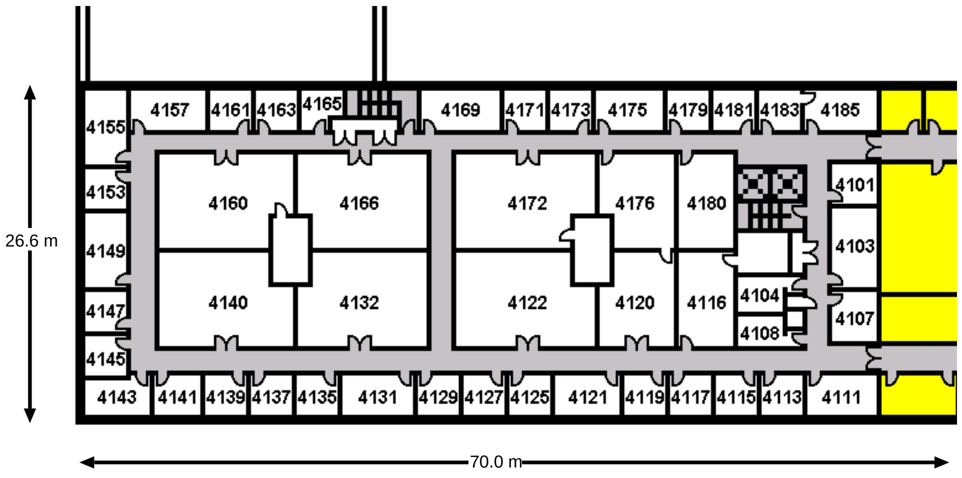

Figure 6: The indoor Rover system is currently implemented in the 4th floor of the A.V. Williams Building at the University of Maryland.

a seamless experience of location-aware computing to clients as they move through the system. With this in mind, we have a set of different projects in both the short and the long term.

- Experiment with a wider range of client devices, specially the ones with limited capabilities. They include devices with low-resolution graphics, limited color choices, or only a few lines of text display area.
- For the more-capable devices, we are experimenting with location-aware streaming video services.
- Integrate different other wireless air interfaces to the Rover system. Bluetooth-based LAN is emerging as an important standard today, and it is a logical next technology to experiment with. In the longer term, we are expecting to interact with cellular providers to define and implement mechanisms that will allow Rover clients to interact over the cellular interface.
- Implement the other different location services. We are currently building custom hardware that will allow the deployment of the PinPoint Technology (see Appendix) for device location. We are also experimenting with other mechanisms for better location estimation.
- Implement the multi-Rover system.
- Campus-wide deployment of Rover. In the near term, we are hoping to deploy a Rover system in the campus of the University of Maryland, College Park. Initially, independent Rover systems will be deployed to serve clients of specific departments. Beyond that these systems will be able to interact using the inter-Rover Controller protocols of a multi-Rover system. The Rover controllers will be co-located with the web servers, and the content management will be handled jointly for both the systems in an integrated manner.

We believe that Rover Technology will greatly enhance the user experience in a large number places, including visits to museums, amusement and theme parks, shopping malls, game fields, offices and business centers. The system has been designed specifically to scale to large user populations. Therefore, we expect the benefits of this system to be higher in such large user population environments.

References

[1] http://www.bluetooth.com.

[2] http://www.irda.org.

[3] J. Agre, D. Akenyemi, L. Ji, R. Masuoka, and P. Thakkar. A Layered Architecture for Location-based Services in Wireless Ad Hoc Networks. In *Proceedings of IEEE Aerospace Conference*, March 2002.

[4] P. Bahl and V.N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *Proceedings of Infocom*, Tel Aviv, Israel, March 2000.

[5] N. Davies, K. Cheverst, K. Mitchell, and A. Efrat. Using and Determining Location in a Context-Sensitive Tour Guide. *IEEE Computer*, 34(8), August 2000.

[6] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins. *GPS: Theory and Practice*. Springer-Verlag, Wein, NY, 1997.

[7] IEEE. Wireless LAN medium access control (MAC) and physical layer (PHY) specification, Standard 802.11, 1999.

[8] J. Mitola. The Software Radio Architecture. *IEEE Communications Magazine*, 5, May 1995.

[9] P. Mockapetris. Domain names - implementation and specification, RFC 1035, November 1987.

[10] C.E. Perkins. IP mobility support, RFC 2002, October 1996.

[11] A.J. Viterbi. *CDMA: Principles of Spread Spectrum Communications*. Addison-Wesley, 1995.

A Location Determination Techniques

The location service in Rover is decoupled from the Rover controller. Both the clients and the Rover controller can directly interact with an external location service to obtain client locations. We outline some of the different technologies that are available to provide client location information. Some of these techniques are applicable to specific clients operating with specific technologies.

- **GPS**: Perhaps the most popular option of obtaining location information today is through the use of GPS receivers. Here the client device interfaces with a GPS receiver; no other infrastructure is required. GPS can provide user location to an accuracy of around 3 meters provided the device has good signal reception from at least four GPS satellites. The main problem with GPS is that it is generally not workable indoors.

- **E-911**: Another option is the E-911 emergency location service that cellular service providers are required by the FCC to provide between October 2001 and December 2005. According to the requirements for this Automatic Location Information (ALI) the cellular system should be able to locate the calling device to within an accuracy of 50 to 300 meters depending on specific systems being used. Here, the client devices would be cell phones and the Rover system would interface with the cellular service provider (for both location and data connectivity).

- **WLAN signal strength**: Another option, motivated by the increasing popularity of wireless LANs indoors, is to determine the position of a client device relative to the base station positions based on the received signal strength from base stations at the client device [4]. [One could also use the received signal strength from the client at the different base stations, but this requires access and modification to the base stations.]

The determination of location information from signal strength is nontrivial because the strength of a received signal depends not only on the distance between the transmitter and receiver but also on obstructions and multipath interference. Nonetheless, we are finding out that the problem is surmountable with appropriate processing of the received signal strengths. In particular, we have developed a technology for IEEE 802.11 WLANs that, when used in the UMD Computer Science Department building, determines client device location to within a meter accuracy.

Better accuracy can be obtained by reducing the cell size, that is, populating the area with more base stations and correspondingly decreasing the transmission power. If this is done to a high degree, for example, using Bluetooth wireless devices and base stations, then merely identifying the closest Bluetooth access point for a client device may be sufficient, without considering the received signal strength. The disadvantage here is the need to deploy many base stations with adequate spatial spread.

- **Signal propagation delays**: An "obvious" way to obtain the distance between two wireless devices (in our case, client device and base station) is to obtain the roundtrip propagation time between them, by having one device send a signal that is echoed back by the other. These techniques can be used indoors and outdoors. In fact, the GPS system is based on this technique, and is available only outdoors due to the line-of-sight requirements. For very accurate location inference using signal propagation delays, measurements need to be made with nanosecond accuracy (corresponding to 30 centimeter error), and this is nontrivial to achieve inexpensively (due to signal waveform deterioration and multipath interference).

  Another approach is to measure the differences between arrival times of a client signal at different base stations, which can then be used to triangulate the client position. A major problem here is the need for the base station clocks to be synchronized to within nanosecond accuracy, which generally requires expensive Caesium clocks.

  We have developed a new technique, called PinPoint Technology, that provides one-way propagation times using only standard crystal oscillator clocks. The technique requires each node to have the capability to measure the arrival time of a received signal according to its *local clock*. The local clocks can be inexpensive clocks with high drift rates (e.g. 0.1 millisecond per second). The local clocks of the devices are not synchronized; indeed, the PinPoint technique also yeilds the current drift attributes needed to synchronize the clocks (until their drift rates change).

- **Manual location entry**: A decidedly low-tech option is to have the user identify his/her location manually, typically by pointing on an interactive map on the client device. In fact, even with one or more of the above automated techniques, this manual option is probably essential to achieve fine-grained resolution inexpensively, say identifying which particular exhibit is in front of the user. Such a fine-tuning through user input is employed in the Lancaster Guide system [5].