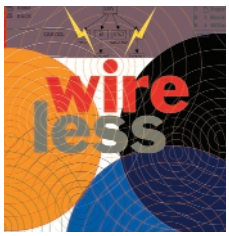
Rover: Scalable Location-Aware Computing



Rover technology adds a user's location to other dimensions of system awareness, such as time, user preferences, and client device capabilities. The software architecture of Rover systems is designed to scale to large user populations.

Suman Banerjee
Sulabh Agarwal
Kevin Kamel
Andrzej Kochut
Christopher Kommareddy
Tamer Nadeem
Pankaj Thakkar
Bao Trinh
Adel Youssef
Moustafa Youssef
Ronald L. Larsen
A. Udaya Shankar
Ashok Agrawala
University of Maryland, College Park

Consider a group touring the museums in Washington, D.C. The group arrives at a registration point, where each person receives a handheld device with audio, video, and wireless communication capabilities—an off-the-shelf PDA available in the market today. A wireless-based system tracks the location of these devices and presents relevant information about displayed objects as the user moves through the museum. Users can query their devices for maps and optimal routes to objects of interest. They can also use the devices to reserve and purchase tickets to museum events later in the day. The group leader can send messages to coordinate group activities.

The part of this system that automatically tailors information and services to a mobile user's location is the basis for *location-aware computing*. This computing paradigm augments the more traditional dimensions of system awareness, such as time-, user-, and device-awareness. All the technology components to realize location-aware computing are available in the marketplace today. What has hindered the widespread deployment of location-based systems is the lack of an integration architecture that scales with user populations.

We developed Rover technology to meet this need.[1] We have completed the initial implementation of a system based on it and have validated its underlying software architecture, which achieves system scalability through fine-resolution, application-specific resource scheduling at the servers and network.

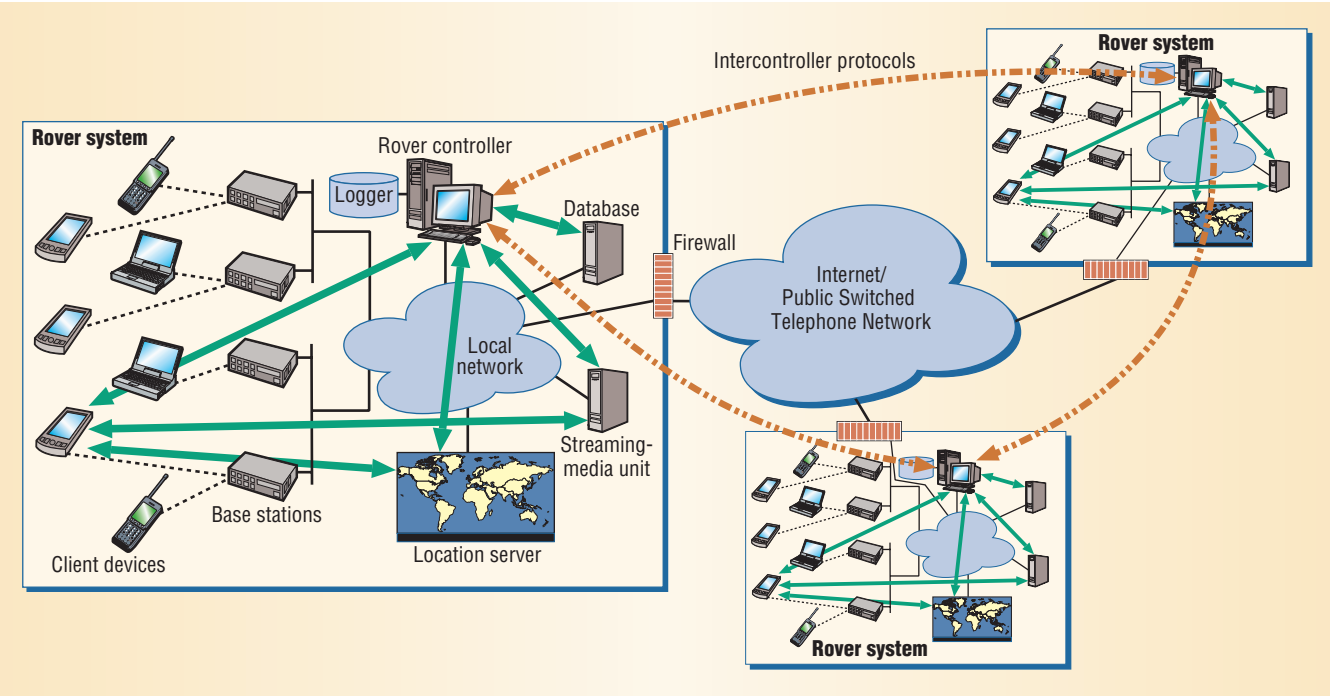
ROVER ARCHITECTURE

Rover technology tracks the location of system users and dynamically configures application-level information to different link-layer technologies and client-device capabilities. A Rover system represents a single domain of administrative control, managed and moderated by a Rover controller. Figure 1 shows a large application domain partitioned into multiple administrative domains, each with its own Rover system—much like the Internet's Domain Name System.[2]

End users interact with the system through Rover client devices—typically wireless handheld units with varying capabilities for processing, memory and storage, graphics and display, and network interfaces. Rover maintains a profile for each device, identifying its capabilities and configuring content accordingly. Rover also maintains end-user profiles, defining specific user interests and serving content tailored to them.

A wireless access infrastructure provides connectivity to the Rover clients. In the current implementation, we have defined a technique to determine location based on certain properties of the wireless access infrastructure. Although Rover can leverage such properties of specific air interfaces,[1] its location management technique is not tied to a particular wireless technology. Moreover, different wireless interfaces can coexist in a single Rover system or in different domains of a multi-Rover system. Software radio technology[3] offers a way to integrate the different interfaces into a single device. This would allow the device to easily roam between different

0018-9162/02/$17.00 © 2002 IEEE



Rover systems, each with different wireless access technologies.

A server system implements and manages Rover's end-user services. The server system consists of five components:

- The *Rover controller* is the system's "brain." It manages the different services that Rover clients request, scheduling and filtering the content according to the current location and the user and device profiles.
- The *location server* is a dedicated unit that manages the client device location services within the Rover system. Alternatively, applications can use an externally available location service, such as the Global Positioning System (GPS).[4]
- The *streaming-media unit* manages audio and video content streamed to clients. Many of today's off-the-shelf streaming-media units can be integrated with the Rover system.
- The *database* stores all content delivered to the Rover clients. It also serves as the stable store for the user and client states that the Rover controller manages.
- The *logger* interacts with all the Rover server components and receives log messages from their instrumentation modules.

The server system exports a set of well-defined interfaces through which it interacts with the heterogeneous world of users and devices. Third-party developers can use these interfaces to develop applications that interact with the system.

For multi-Rover systems, 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.

ROVER SERVICES

Rover offers two kinds of services to its users. We refer to them as *basic data services* and *transactional services*.

- Basic data services use text, graphics, audio, and video formats. Users can subscribe dynamically to specific data components through the device user interface. Rover filters the available media formats according to the device's capabilities. The basic data service involves primarily one-way interaction.
- Transactional services have commit semantics that require coordinating state between the clients and Rover servers. E-commerce interactions are examples of this service class.

Location is an important attribute of all objects in Rover. Several techniques exist for estimating an object's location, including the GPS and radio-frequency techniques based on signal strength or signal propagation delays. The choice of technique significantly affects the granularity and accuracy of the location information. Rover therefore uses a tuple of *value*, *error*, and *time stamp* to identify an object's location.[1] The value is an estimate of the object's location (either absolute or relative to some well-known location). The error identifies the uncertainty in the estimate. The time stamp identifies when the estimate was made.

The accuracy required of location information depends on the context of its use. For example, an

Figure 1. Rover physical architecture in the context of a multi-Rover system. The Rover controller manages system services, including communication between individual Rover systems.

A task-scheduling architecture handles the large volume of real-time requests that users generate.

accuracy of 4 meters is adequate to provide walking directions from the user's current location to another location about 500 meters away. However, it is inadequate for locating a particular painting on a museum wall directly in front of the user. Obviously, the accuracy of location information improves significantly with direct user input. For example, the user can directly input a location on a map displayed on his or her device.

Rover includes support for operations to filter, zoom, and translate map display. Filter operations are keyed to a set of attributes that identify certain properties of map objects. Rover generates the filters based on the user's context. The filters select and map the appropriate object subset for display. For example, one user may be interested only in the restaurants in a specific area, while another wants to view only museum and exhibition locations.

A displayed map's zoom level identifies its granularity. The user's context, consisting of a location and profile, determines its default setting. For example, inside a museum, the map shows a detailed museum layout. When the user steps outside, the display zooms out to an area map with points of interest in the geographic vicinity. Additionally, the user can choose to alter the major zoom level through explicit input. In either case, appropriate filters are used to selectively display different objects on a map at any zoom level. Rover automatically translates the map displayed on the client device as the user moves to a new region.

SYSTEM SCALABILITY

Two potential bottlenecks can hinder the system's scalability. One is the server system, which must handle a large number of client requests with tight real-time constraints. The other is the wireless access points, which have limited bandwidth.

To handle the large volume of real-time requests that users generated, we developed the *action model*, a fine-grained, real-time, application-specific architecture that allows Rover systems to scale to large user populations.

To make its implementation more efficient, we divided the Rover server components into two classes based on the user request volumes they handle:

- primary servers directly communicate with the clients and therefore directly handle large volumes of user requests. They include the Rover controller, location server, and streaming media unit; and
- secondary servers, which communicate only with primary servers to provide back-end system capabilities, include the Rover database and logger.

Only the primary servers need to implement the Action model.

Action model

The Action model avoids the overhead of thread context switches and allows more efficient scheduling of execution tasks. The Rover controller implements this architecture.

In the action model, scheduling occurs in "atomic" units called actions. An action is a small piece of code that has no intervening I/O operations: Once an action begins execution, another action cannot preempt it. Consequently, given a specific server platform, it is easy to accurately bound an action's execution time.

A *server operation* is a transaction, either client- or administrator-initiated, that interacts with the Rover controller: Examples in the museum scenario would include registerDevice, getRoute, and locateUser. A server operation consists of a sequence—or more precisely, a partial order—of actions interleaved with asynchronous I/O events. Each server operation has one action for each kind of I/O event response.

A server operation at any given time has zero or more actions eligible for execution and is in one of three states:

- ready-to-run—at least one of the server operation's actions is eligible for execution but none is executing;
- running—one action is executing (in a multiprocessor setup, several of the operation's actions can execute simultaneously); or
- blocked—the server operation is waiting for an asynchronous I/O response, and no actions are eligible to be executed.

An *action controller* uses administrator-defined policies to decide the execution order of the eligible actions. The scheduling policy can be simple and static, such as priorities assigned to server operations. It also can be time based, such as earliest-deadline-first or a function of real-time costs. In any case, the controller picks an eligible action and executes it to completion, then repeats the process—waiting only if there are no eligible actions, presumably because all server operations are waiting for I/O completions.

A simple action API defines the management and execution of actions:

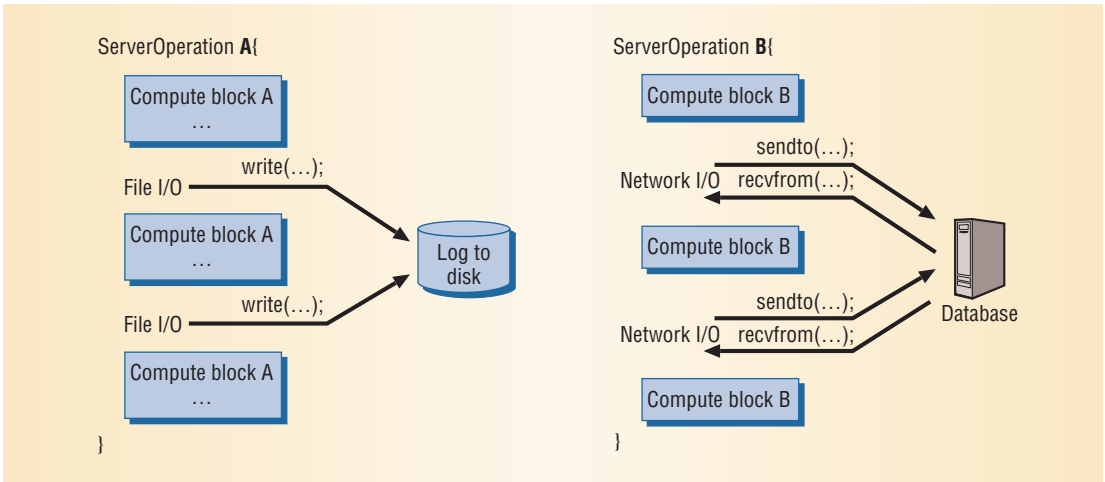


Figure 2. Server operations used in the experimental evaluation of the action model. Scenario A (left) interleaves computation with file-write operations. Scenario B (right) interleaves computation with network I/O interactions.

- `init (action id, function ptr)`—initializes a new action (identified by `action id`) for a server operation. `Function ptr` identifies the function (or piece of code) associated with the action.
- `run (action id, function parameters, deadline, deadline failed handler ptr)`—marks 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 execute; the deadline is soft—that is, violating it leads to some penalty but not system failure. If the action controller cannot 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)`—cancels a ready-to-run action, provided it is not executing. `Cancel handler ptr` indicates a cleanup function; it can be null.

Actions versus threads

There are several ways to implement the Rover controller using a thread model. For example, each server operation could have a separate thread, or each user could have a separate thread handling all its operations. Both of these approaches 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.

This approach reduces the thread-switching overhead, but there are drawbacks. For one, the threads package restricts the ability to optimize scheduling, especially in time-based scheduling. More importantly, each operator thread executes its set of operations in sequence, which severely limits the 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 essentially re-creates the action model within each thread.

We compared the performance of action-based versus more traditional thread-based systems in two kinds of server operations, shown in Figure 2:

- Scenario A—a computation-intensive scenario with 10,000 *processor-bound server operations*, in which each server operation has three compute blocks, interleaved with two file-write operations. In each of these server operations, the second and third I/O compute blocks need not await completion of the prior file I/O write operation.
- Scenario B—an I/O-intensive scenario with 100 *I/O-bound server operations*, in which each server operation has three compute blocks, interleaved with two network I/O operations. In each of these server operations, the second and third compute blocks can start only after the prior network I/O operation finishes. We use UDP to implement network I/O interaction. Since our focus is on the comparison of action-based versus thread-based systems, we avoid issues of packet loss and retransmissions by considering only those experiments in which no UDP packets were lost in the network.

We used two execution platforms: M1 and M2. M1 runs Linux on a 600-MHz Intel Pentium III processor with 96 Mbytes of RAM. M2 runs Solaris on a Sun Ultra 5 with a 333-MHz Sparc processor and 128 Mbytes of RAM. For the thread-based implementation, we used the LinuxThreads library for the M1 platform and the Pthreads library for the M2 platform; both are implementations of the Posix 1003.1c threads package.

Table 1. Comparisons of overheads for action-based and thread-based systems (in milliseconds).

Figure 2 scenario	Machine specifications	Action-based	Thread-based (threads used) 1	5	10	50	100
A	M1: Pentium/Linux	24.27	299.36	299.93	300.46	304.50	310.31
A	M2: Sparc/Solaris	62.82	1,000.90	1,012.54	1,041.60	1,012.83	1,031.25
B	M1 controller, M2 database	11.61	3,711.94	1,302.20	1,011.49	893.10	728.30

The 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 and 35 ms.

In the action-based scenarios, we implemented each compute block as a separate action. In the thread-based scenarios, we experimented with different numbers of threads, executing each thread an equal number of server operations for perfect load balancing between the different threads.

Table 1 shows the overheads obtained in each case, where overhead is the total execution time minus the fixed, identical, and unavoidable computation and communication costs for the two scenarios. The results represent the mean execution overheads of 10,000 server operations in scenario A and 100 server operations in scenario B, which were required to obtain low variance. The computation-intensive server operations show little performance gain in trying to overlap computation with file I/O communication—not enough to justify the overhead of a multithreaded implementation. Among the thread-based implementations, a thread-based system with a single thread performs best.

For the I/O-intensive server operations, a multithreaded implementation is useful because computation and communication can overlap. Consequently, the best performance for the thread-based system occurs with the maximum number of threads—specifically, one thread for each server operation.

However, as Table 1 shows, the action-based implementation in both scenarios has about an order of magnitude lower overhead compared with the best thread-based implementation.

COMMUNICATION INTERFACES

For the wireless interface to client devices, we considered two link-layer technologies: IEEE 802.11 and Bluetooth. Bluetooth is power efficient and therefore better at conserving client battery power. According to current standards, Bluetooth can provide bandwidths up to 2 Mbps. In contrast, IEEE 802.11 is less power efficient but widely deployed and currently provides bandwidths up to 11 Mbps. In areas where these high-bandwidth alternatives are not available, Rover client devices will use the lower bandwidth interfaces that cellular wireless technologies provide.

Figure 3 shows how Rover's controller interacts with other parts of the system and with the external world. The controller uses the *location interface* to query the location service about the

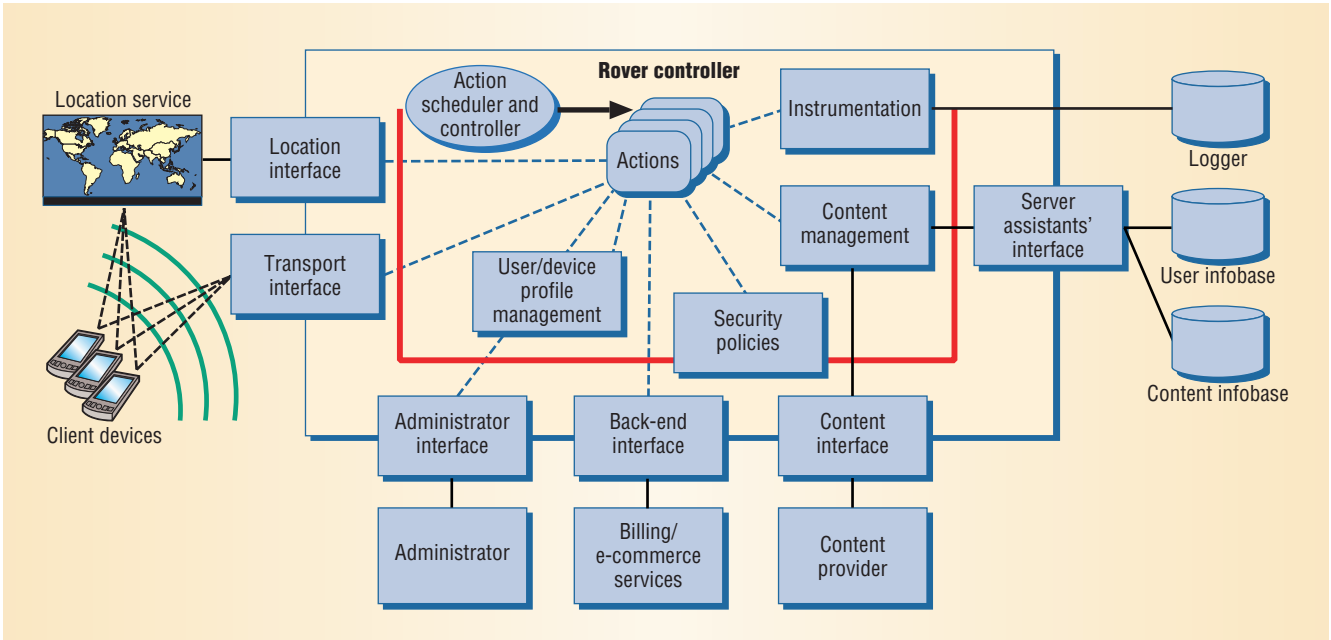


Figure 3. Rover's logical architecture. The Rover controller interacts with the external world through interfaces for location, transport, administration, back-end services, content, and secondary server assistants.

positions of client devices and the *transport interface* to identify data formats and interaction protocols for communicating with the clients. It uses the *server assistants' interface* to interact with secondary servers like the database and the streaming-media unit and the back-end interface to interact with external services, such as credit card authorization for e-commerce purchases. Third-party providers typically offer these external services.

System administrators can use the *admin interface* to oversee the Rover system, including monitoring the Rover controller, querying client devices, updating security policies, issuing system-specific commands, and so on.

The *content interface* lets content providers update the information and services that the Rover controller serves to client devices. Having a separate content interface decouples the data from the control path.

INITIAL IMPLEMENTATION

We have successfully built Rover prototype systems and tested them in both indoor and outdoor environments at the University of Maryland, College Park. A preliminary Windows-based test implementation ran Windows 2000 for the controller and Windows CE for the client devices. However, the current implementation runs under Linux.

We implemented the Rover controller on an Intel Pentium machine running Red Hat Linux 7.1 and the clients on Compaq iPAQ model H3650 Pocket PCs running Familiar's Linux distributions for PDAs (http://familiar.handhelds.org). Wireless access is over IEEE 802.11 wireless LANs. Each Compaq iPAQ includes a wireless card that attaches to the device through an expansion sleeve.

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

For our outdoor experiments, we interfaced a Garmin e-Trex (http://www.garmin.com/products/etrex/) GPS device to the Compaq iPAQs and obtained device location accuracy between 3 and 4 meters. Figure 4 shows the iPAQ Rover client's default display, which marks different user locations as dots on an area map.

We implemented the indoor Rover system in a 26.6 × 70-meter area on the fourth floor of the Computer Science Department building. In this implementation, the location service uses signal-strength measurements from different base stations. About 12 base stations are distributed all over the building, and the client device can typically receive beacons from five or six of them. We get an accuracy of about 2 meters in this environment, using very simple signal-strength estimation techniques.

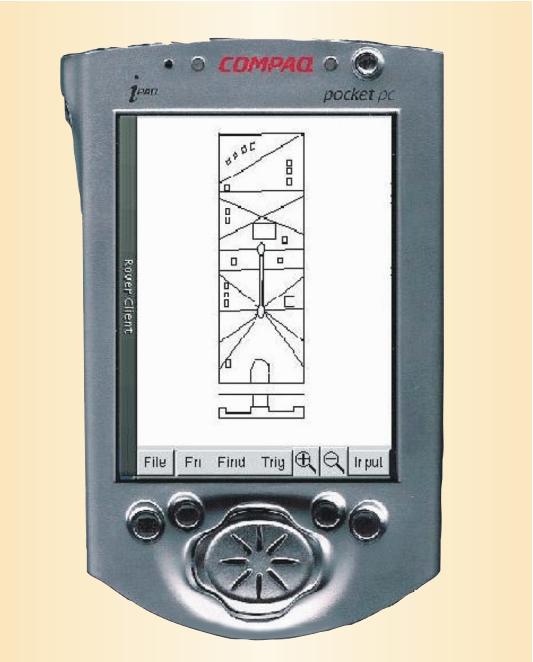


Figure 4. Rover client default display for Compaq iPAQ marks user locations as dots on an area map.

In both these environments, we implemented the basic Rover system functionality, which included

- user activation and deactivation procedures;
- device registration and deregistration procedures;
- periodic broadcast of events of interest from the Rover controller to the users in specific locations;
- unicasts from the controller according to user-specified time, location, or context-dependent conditions;
- both simple-text-messaging and voice-chat interaction between users; and
- an administrator's console, allowing a global view of all system users and their locations.

The administrator can directly interact with all users or a specific subset based on location or other user attributes. Users have the option of making their location visible to other users.

Rover is currently available as a deployable system using specified technologies, both indoors and outdoors. Ultimately, our goal is to provide a completely integrated system that uses different technologies and allows a seamless experience of location-aware computing to clients as they move through the system. With this in mind, we have various short- and long-term projects:

- Experiment with a wider range of client devices, both those with limited text and graphics display capabilities and those that can

support richer functionality, such as location-aware streaming video services.
- Integrate more wireless air interfaces with the Rover system. Bluetooth is a logical next technology to experiment with. In the longer term, we expect to work with cellular providers to define and implement mechanisms that will let Rover clients interact over the cellular interface.
- Implement more location-determination techniques. We are experimenting with new mechanisms for better location estimation, including a signal propagation delay-based technique, called PinPoint Technology, developed at the University of Maryland.
- Implement the multi-Rover system.
- Deploy Rover campus-wide at the University of Maryland, College Park. Initially, we expect to deploy independent Rover systems to serve clients of specific departments. These systems will subsequently interact using the inter-Rover controller protocols of a multi-Rover system. We will colocate the Rover controllers with the Web servers and integrate the content management for both systems.

We believe that Rover technology will greatly enhance the user experience in many places, including museums, amusement and theme parks, shopping malls, game fields, offices, and business centers. We designed the system specifically to scale to large user populations and expect its benefits to increase with them. ■

References

1. S. Banerjee et al., *Rover Technology: Enabling Scalable Location-Aware Computing*, tech. reports UMIACS-TR 2001-89 and CS-TR 4312, Dept. Computer Science, Univ. of Maryland, College Park, Md., Dec. 2001.
2. P. Mockapetris, "Domain Names: Implementation and Specification," Internet Engineering Task Force, RFC 1035 (Internet standard), Nov. 1987; http://www.ietf.org/rfc/rfc1035.txt.
3. J. Mitola, "The Software Radio Architecture," *IEEE Comm.*, vol. 5, May 1995, pp. 26-38.
4. B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins, *GPS: Theory and Practice*, Springer-Verlag, Wien, N.Y., 1997.

Suman Banerjee *is a PhD candidate in the Department of Computer Science at the University of Maryland, College Park. He received an MS in computer science from the University of Maryland, and a BTech in computer science and engineering from the India Institute of Technology in Kanpur. He is a student member of the IEEE. Contact him at suman@cs.umd.edu.*

Sulabh Agarwal *is a software developer with Epic Systems Corp., Madison, Wisconsin. He received an MS in computer science from the University of Maryland, College Park. He received a BTech from the India Institute of Technology, Delhi.*

Kevin Kamel *is a faculty research assistant at the University of Maryland Institute for Advanced Computer Studies, College Park. He received a BS in computer engineering from the University of Maryland, College Park. Contact him at kamelkev@umiacs.umd.edu.*

Andrzej Kochut *is a PhD student in computer science at the University of Maryland, College Park. He received an MS in computer science from the University of Warsaw, Poland. Contact him at kochut@cs.umd.edu.*

Christopher Kommareddy *is a PhD student in the Department of Electrical and Computer Engineering, University of Maryland, College Park, where he received an MS in electrical and computer engineering. Contact him at kcr@cs.umd.edu.*

Tamer Nadeem *is a PhD student in computer science at the University of Maryland, College Park. He received an MS in computer science from the University of Maryland, and an MS and a BS in computer science from Alexandria University, Egypt. He is a student member of the IEEE and the ACM. Contact him at nadeem@cs.umd.edu.*

Pankaj Thakkar *is a software engineer with AskJeeves. He received an MS in computer science from the University of Maryland, College Park, and a BTech in computer science and engineering from the India Institute of Technology, Delhi.*

Bao Trinh *is a faculty research assistant in the Department of Computer Science at the University of Maryland, College Park, where he received an MS in computer science. Contact him at bao@mindlab.umd.edu.*

Adel Youssef *is a PhD student in computer science at the University of Maryland, College Park. He received an MS in computer science from the Uni-*

versity of Maryland, College Park, and an MS and BS in computer science from Alexandria University, Egypt. Contact him at adel@cs.umd.edu.

***Moustafa Youssef** is a PhD candidate in computer science at the University of Maryland, College Park. He received an MS in computer science from the University of Maryland, and an MS and a BS in computer science from Alexandria University, Egypt. He is a student member of the IEEE, the IEEE Computer Society, and the IEEE Communication Society. Contact him at moustafa@cs.umd.edu.*

***Ronald L. Larsen** is dean of the School of Information Sciences at the University of Pittsburgh. He received a PhD in computer science from the University of Maryland. He is a member of the IEEE, the IEEE Computer Society, and the ACM.*

***A. Udaya Shankar** is a professor in the computer science department and the Institute for Advanced Computer Studies at the University of Maryland, College Park. He received a PhD in electrical engineering from the University of Texas at Austin. Contact him at shankar@cs.umd.edu.*

***Ashok Agrawala** is a professor in the Department of Computer Science at the University of Maryland and holds joint positions with the University of Maryland Institute for Advanced Computer Studies (UMIACS) and the Department of Electrical Engineering. He received an MS and a PhD in applied mathematics from Harvard University and an ME and a BE in electrical engineering from the Indian Institute of Sciences, Bangalore. He is a Fellow of the IEEE and a member of the ACM, AAAS, and Sigma Xi. Contact him at agrawala@cs.umd.edu.*