5-22-2001

Infrastructure for Distributed Applications in Ad Hoc Networks of Small Mobile Wireless Devices

Alan Kaminsky
Rochester Institute of Technology

Infrastructure for Distributed Applications in Ad Hoc Networks of Small Mobile Wireless Devices

Alan Kaminsky
Information Technology Laboratory
Rochester Institute of Technology
ark@it.rit.edu

Abstract. Mobile wireless computing devices such as cellphones, pagers, personal digital assistants, pocket PCs, and tablet computers are all potential platforms for participating in small group, wireless, many-to-many distributed applications. The networking technology needed to support such applications is readily available. However, almost all existing middleware infrastructure for distributed applications was designed for central servers and wired connections. The Anhinga Infrastructure described here runs entirely on the wireless mobile devices and so does not require any central server support. The Anhinga Infrastructure provides a message broadcast ad hoc networking protocol and a distributed computing platform based on lightweight versions of Java, Jini Network Technology, and tuple spaces.

1. Introduction

1.1. The Emerging Computing Milieu

Computers are changing radically and rapidly. Even while computer processing power and memory capacity continue to double every 18 months according to Moore's Law, computers continue to evolve to smaller and smaller sizes, from mainframes to minicomputers to desktops to laptops to palmtops. As their size has decreased, computers have become increasingly portable, evolving from sessile* systems to mobile devices. With mobility has come the need to sever the computers' hardwired network connections, replacing them with wireless connections.

Within the next few years, the most prevalent computers will actually be small mobile wireless devices such as cellphones, pagers, personal information managers (PIMs), personal digital assistants (PDAs), pocket-sized PCs, and small tablets, as well as devices which combine several of these functions. Their small size, low cost, and wide availability from many manufacturers will ensure that many, many people will have one (or more than one).

The computing environment defined by these small mobile wireless devices is vastly different from today's predominant desktop computing environment. Each individual device's computing power is limited compared to a traditional personal computer. They use slower processors. Their memory sizes range from a few hundred kilobytes in the smallest devices to a few megabytes in the larger devices. Their wireless connections typically run at less than one megabit per second. Many of these limitations arise from the need to get reasonable service times out of the devices' batteries.

Unlike desktop computers in which one operating environment is preeminent (Microsoft Windows on Intel-compatible processors), small mobile wireless devices' operating environments are much more diverse, since no one manufacturer dominates the market. They use a variety of processors and special-purpose operating systems and are programmed in a variety of languages. Any randomly-assembled collection of small mobile wireless devices can be expected to present a highly heterogeneous computing environment.

* **sessile** *adj* . . . 2 : permanently attached or established : not free to move about [*Webster's New Collegiate Dictionary*]

Finally, unlike computer systems with hard-wired network connections and fixed network configurations, small mobile wireless devices are constantly coming into and going out of contact with new devices as their users move about. Each user has a "Wireless Personal Area Network" consisting of the ever-changing set of other devices within wireless range of his or her own device. From the user's point of view, other devices will "come up" and "go down" much more frequently than computer systems on a hard-wired network do.

In the small mobile wireless device computing environment, it will not be possible to run large, complex operating systems and applications as is typically done in the desktop computing environment. The devices are too small to run that kind of software. Instead, *distributed* applications, which gain their capabilities from collections of small programs runnings in separate devices and working in concert, will be necessary.

1.2. Unfulfilled Potential

Small mobile wireless computing devices have vast potential for use in distributed applications — potential that is, however, largely unfulfilled for two reasons: the nature of the existing computing infrastructure, and the lack of needed technology.

The existing computing infrastructure is built from sessile resources: server machines of all kinds (web servers, email servers, file servers, database servers, application servers, DNS servers, DHCP servers, and so on), client machines of all kinds (PCs, workstations, and so on), and a wired network with all its cables, hubs, switches, and routers. This infrastructure superbly supports client-server architectures, where the application's central data and logic reside in a server machine and the application's human interfaces reside in the users' client machines.

Small mobile wireless devices are being fitted into the existing infrastructure by treating them as additional client devices with wireless connections to the wired network. Cellphones and PDAs with wireless modems, for example, are used to send and receive email and instant messages.

From the perspective of realizing the full potential of small mobile wireless devices, the existing infrastructure and client-server architectures have drawbacks. If the server in a client-server application becomes unavailable, the application is unusable. If wireless devices are in a place where wireless access to the wired network is out of range or not installed, the devices will not be able to communicate with each other via applications that presume a central server, even though the devices are perfectly capable of communicating directly with each other.

In addition to traditional client-server architectures, the computing power available in small mobile wireless devices opens up the possibility of distributed architectures involving *peer-to-peer* interactions among the devices themselves. In such an architecture, a wired network is not a necessity. Using their wireless networking capabilities, the devices could spontaneously discover each others' existence, connect to each other to form an *ad hoc* network, and communicate and collaborate with each other even if there is no portal to the wired network nearby.

The low-level technology needed to make ad hoc peer-to-peer mobile distributed architectures a reality is already available. Wireless local area networking technology, including IEEE 802.11 and Bluetooth [1], is now available. Platform-independent programs, which are needed to run distributed applications in an environment of heterogeneous mobile devices, can be developed using the Java programming language [2]. The Java 2 Micro Edition (J2ME) [3], [4], [5] is a version of Java designed to run in small devices with limited memory. Several J2ME-enabled cellphones, pagers, and PDAs are available, and more J2ME-enabled devices are on the way. Java APIs for Bluetooth are under development [6].

However, the high-level technology needed to make ad hoc peer-to-peer mobile distributed architectures a reality is still nascent. Routing protocol standards for networks of mobile devices — such as those the IETF's MobileIP [7] and MANET [8] Working Groups are designing — are still under development. Distributed middleware standards — such as CORBA [9] and Jini Network Technology [10] — were designed for sessile hosts and wired

networks, and software implementations of these standards are simply too large to use in small mobile wireless devices. There is a lack of distributed middleware designed specifically for ad hoc networks of small mobile wireless devices.

1.3. The Anhinga Project

To provide this middleware, the Anhinga Project [11] is developing a distributed communication and collaboration infrastructure specifically designed for ad hoc networks of small mobile wireless devices. This paper describes the architecture of the Anhinga infrastructure and the network protocols it uses.

Section 2 lists the kinds of distributed applications we envision will use the Anhinga infrastructure. These applications are intended to run on groups of *proximal* wireless devices and are characterized by *many-to-many* communication patterns. Section 3 describes the Many-to-Many Protocol (M2MP), a network protocol based on broadcast messages that is designed for many-to-many communication among proximal devices. Section 4 describes the architecture of the Anhinga infrastructure, which is based on variations of J2ME, Jini, and tuple spaces. Section 5 concludes with the Anhinga Project's current status and future plans.

2. Distributed Applications

2.1. Examples of Distributed Applications

Potential applications for a distributed communication and collaboration infrastructure built using small mobile wireless devices include:

Device-to-device chat and instant messaging. Persons use their devices to carry out nonvocal conversations in settings where vocal communication is difficult or impossible, such as conversations in quiet spaces, conversations in noisy spaces, and conversations between hearing-impaired and hearing persons. Conversations are carried out in real time among a group of persons ("chat") or, as a special case, between just two persons ("instant messaging" or IM). In quiet settings such as a library, lecture, or meeting, persons could carry out silent conversations without disturbing others. In noisy settings such as public transit, airfield tarmacs, or large gatherings, persons could carry out conversations without having to raise their voices to overcome the noisy environment. Hearing-impaired persons and hearing persons also could converse via their devices. This is especially attractive in educational settings to facilitate interaction between hearing-impaired and hearing students and instructors when an interpreter is not available. Another application for device-to-device chat is for students to ask the instructor questions in a lecture. In a large lecture hall, using a device to contact the instructor may be more effective than raising one's hand. Also, if the application lets questions be asked anonymously (or under a pseudonym), students may be encouraged to ask questions they would not have asked if they had to raise their hands and identify themselves.

Collaborative groupware. Persons coming together for a face-to-face meeting use their devices for:

- *Whiteboard.* Whatever any person types or draws on his or her device, all the other persons see on their devices.
- *Collaborative authoring.* All persons can simultaneously edit a document. Each person, on his or her device, sees the results of all persons' inputs in real time. The final document is available on all devices.
- *Note taking.* Notes from a meeting can be entered by any person and are saved in a common transcript that is available on all devices.
- *Scheduling.* The date and time of the next meeting are determined by querying all the persons' calendars on their devices, finding a time when everyone is free, and updating all the devices with the chosen time.
- *Information transfer.* Any person can transfer information (files, contacts, and so on) from his or her device to all the other devices.

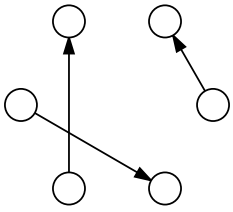
2.2. Characteristics of Distributed Applications

Distributed applications such as the above share several common characteristics:

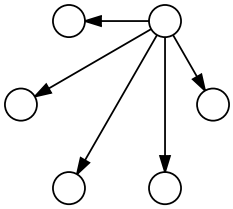
- *Ad hoc networking.* The devices must discover each others' existence, discover each others' ability to run the distributed applications, and

set up network connections automatically, without requiring pre-specified, manual configuration.

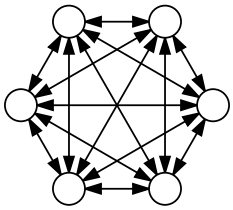
- *No sessile infrastructure.* The distributed applications cannot rely on central servers or connections to the fixed network, since those resources may be unavailable or out of range in the locations where the applications are used. The only infrastructure guaranteed to be available is that provided by the devices themselves and their wireless interconnections.
- *Proximal devices.* The devices are all within close proximity of each other. Chat conversations, groupware collaboration, and so on are taking place among people around the same table or in the same room.
- *Many-to-many communication.* At the application level, every device needs to talk to every other device: every person's chat messages are displayed on every person's devices; every person's calendar on every person's device is queried and updated with the next meeting time. Unlike applications such as web browsing (one-to-one communication) or streaming multimedia (one-to-many communication), the distributed applications envisioned here exhibit *many-to-many* communication patterns (Figure 1).



(a) One-to-one communication, e.g. web browsing



(b) One-to-many communication, e.g. streaming multimedia



(c) Many-to-many communication, e.g. chat, groupware

Figure 1. Communication patterns in distributed applications

3. Many-to-Many Protocol

3.1. Motivation for M2MP

Existing network protocols such as TCP and IP were designed assuming sessile hosts and a fixed network infrastructure. Furthermore, they were designed primarily to handle one-to-one communication patterns, such that each packet is routed from a single source host to a single destination host. More recent protocols, such as Destination Sequenced Distance Vector (DSDV) [12], Dynamic Source Routing (DSR) [13], Ad Hoc On-Demand Distance Vector (AODV) [14], Zone Routing Protocol (ZRP) [15], and Source Tree Adaptive Routing (STAR) [16], extend IP packet routing to ad hoc wireless networks and mobile hosts. However, extending IP-style packet routing into the ad hoc wireless arena leads to complications. Each device must keep track of other devices' addresses and recompute routes as devices enter, move around, and leave the ad hoc network. This in turn causes the devices to consume limited wireless bandwidth and scarce battery power for network maintenance. Some ad hoc network routing algorithms attempt to minimize bandwidth and battery usage by searching for routes only when messages have to be sent between devices (on-demand routing), but this yields no savings in a distributed application where every device must communicate with every other device. Also, the memory footprint of the code and data needed to run these algorithms is problematic for small devices with limited memory sizes.

However, the kinds of distributed applications described in Section 2 don't need IP-style packet routing. Since every device must receive and process every message, there's no need for devices to have addresses, and there's no need to route packets between specific addresses. Instead, messages are broadcast to all devices. While a message may need to identify its source or destination for purposes of

the application, this is an application-level identifier that is not used at the network communication level.

For example, consider a group of proximal mobile wireless devices, those of the students attending a lecture, all running a chat application which lets the students converse silently. Whenever a student types something, the student's device broadcasts a message like this:

```
<chat>
  <session>CS-101 Lecture</session>
  <sender>John Doe</sender>
  <text>Anyone ready for lunch?</text>
</chat>
```

All the other devices receive this message and, seeing that it is a "`<chat>`" message for the chat session they're participating in, display the sender's name and text in the chat log. The address, IP or otherwise, of the device that sent this message makes no difference to the chat application. The messages are processed based on *what they say* (their contents) rather than *where they're sent* from or to (their addresses).

3.2. Design of M2MP

Generalizing these notions, the Many-to-Many Protocol (M2MP) has been designed for use by distributed applications with many-to-many communication patterns running on ad hoc networks of proximal mobile wireless devices. The characteristics of and rationale for M2MP are as follows:

M2MP messages are broadcast to all devices. Wireless transmissions (or at least, wireless *radio* transmissions) are inherently broadcast within a certain proximal area. Thus, every device receives every other device's transmissions at the physical level, and no extra work is needed to broadcast messages, within the proximal group. If the devices participating in a distributed application are farther apart, such that not every device is in range of every other device, M2MP messages can still be broadcast throughout the group by relaying messages from device to device. However, the relaying may be achievable by simple flooding without requiring complicated point-to-point routing algorithms; this is an area for research (see Section 5).

There are no device addresses. Consequently, mobile devices can enter and leave the network in an ad hoc fashion without having to maintain routing tables or employ complicated routing algorithms. To participate in M2MP, a device simply starts listening for and sending broadcast messages. If a certain application needs to distinguish among devices, application-level identifiers are included in the message contents.

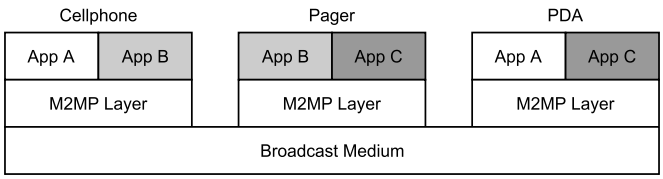


Figure 2. M2MP protocol architecture

Message delivery is mostly reliable but not totally reliable. Since the devices are assumed to be close by, transmission errors and packet losses are assumed to be rare, even with wireless interconnections. Thus, packets don't have to be acknowledged or retransmitted at the network level, which is problematic for broadcast packets. If a particular application requires end-to-end reliable message delivery, that can be provided in a layer above the network protocol; the network protocol shouldn't be expected to provide end-to-end guarantees [17].

Messages may be delivered to different devices in different orders. Because of the possibility of occasional packet loss or varying delays if packets have to be relayed between devices, messages may arrive at different devices in different orders at the network level. The protocol will be simpler (and thus occupy a smaller code footprint in limited-memory devices) if it can simply deliver messages to the application in the order in which the messages arrive at each device without providing global ordering guarantees such as FIFO or causal delivery, which are of limited usefulness at the network level anyway [17]. Again, if a particular application requires global message ordering guarantees, that can be provided in a layer above the network protocol.

Figure 2 shows the M2MP protocol architecture. On each device, one or more distributed applications are layered on top of M2MP, which in turn is layered on top of a broadcast medium. Each application on each device registers with its own M2MP

layer to receive messages whose contents match what the application is looking for. The application specifies one or more *message prefixes* to be matched, such as "`<chat>`" for a chat application. When an application running on some device sends a message via M2MP, M2MP breaks the message into packets and broadcasts each packet via the broadcast medium. Every device receives each packet (barring failures). If a packet's contents show that it is part of a message for which an application on some receiving device has registered an interest — that is, the message's initial bytes match those of some registered message prefix — the receiving device's M2MP layer reassembles the original message from the packets and passes the message to the application. Otherwise, the M2MP layer ignores the packet. If a failure occurs, such as a lost packet (detected by a timeout), the M2MP layer abandons the message and signals an exception to the application.

3.3. Implementing a Broadcast Medium

It remains to describe how to implement a broadcast medium for a group of proximal devices. This depends on the underlying network technology. In the case of Ethernet, including wired Ethernet and wireless Ethernet (IEEE 802.11), implementing a broadcast medium is straightforward: Simply send all M2MP packets to a well-known Ethernet group address. All the M2MP devices instruct their Ethernet interfaces to receive frames from that group address. M2MP can also be operated on sessile hosts with TCP/IP connectivity by sending M2MP packets as User Datagram Protocol (UDP) datagrams to a well-known IP multicast address, such as an unassigned address in the IPv4 local scope of the administratively scoped IPv4 multicast space (239.255.0.0 to 239.255.255.255) [18].

It may also be possible to implement a broadcast medium on top of Bluetooth; this is an area of research (see Section 5). Since Bluetooth was primarily designed for point-to-point communications in cable replacement scenarios, there are two barriers to implementing a broadcast medium. First, the Bluetooth link-level protocol is a half-duplex master-slave protocol. Each "piconet" of communicating Bluetooth devices has a master device, and all

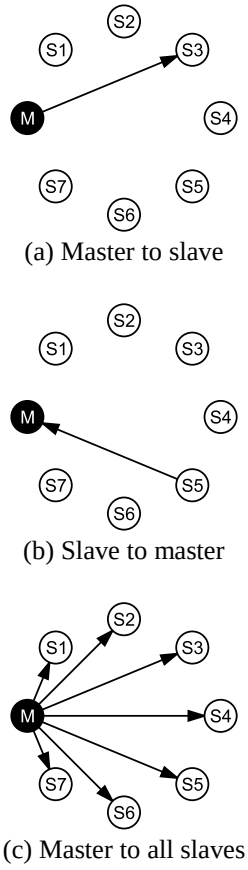
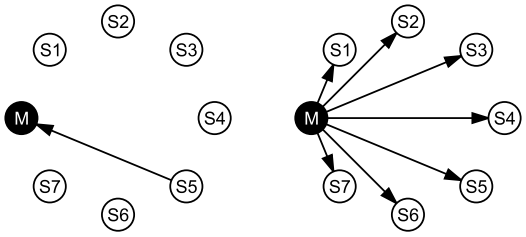


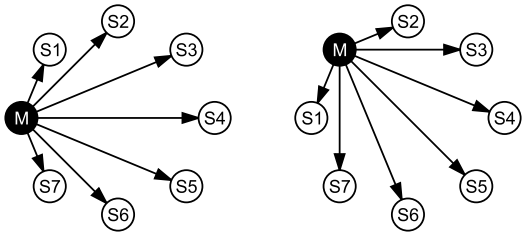
Figure 3. Communication patterns supported in the Bluetooth link-level protocol

other devices in the piconet are slave devices (see Figure 3). The master can send data to a slave, a slave can send data to the master, and the master can broadcast data to all the slaves; but no other communication patterns are possible at the link level. In particular, a slave cannot send data to another slave. Second, a Bluetooth piconet can consist of at most eight active devices, one master and seven slaves. A piconet can include additional "parked" devices, but these devices cannot send or receive data.

To provide communication from any device to any other device in a Bluetooth piconet, a broadcast layer could be added on top of the link-level protocol and below M2MP (see Figure 4). In the broadcast layer, the master device could relay messages from slave to slave, or master status could be rotated among the devices in a round robin fashion, allowing each device to broadcast to the others in



(a) Master relays from one device to all devices



(b) Master status circulates among all devices

Figure 4. Possible ways to implement a broadcast medium on top of Bluetooth

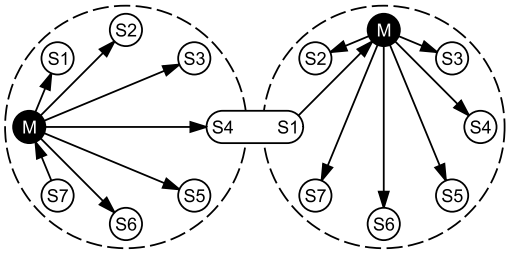


Figure 5. Possible way to form an ad hoc network of more than eight devices on top of Bluetooth

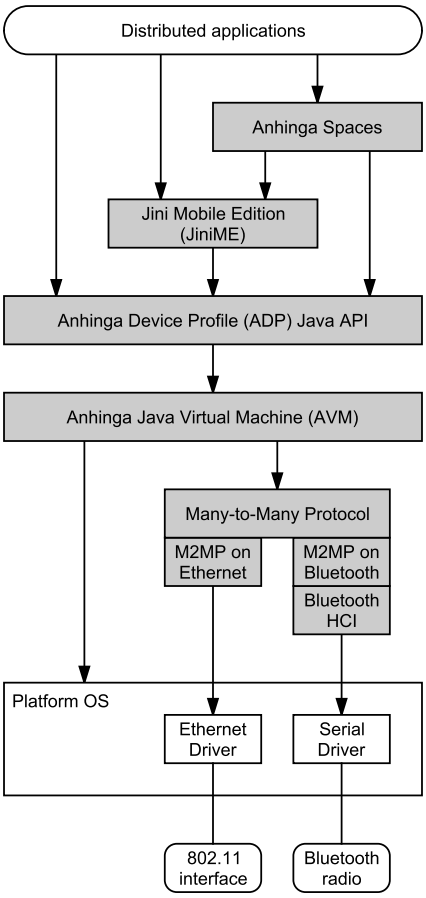


Figure 6. Architecture of the Anhinga Infrastructure

turn. To allow the broadcast medium to scale up beyond eight devices, individual pairs of piconets could be joined by having a certain device active in both piconets at once (known as a "scatternet" in Bluetooth parlance), and the broadcast layer would relay messages from piconet to piconet via the shared devices (see Figure 5).

4. Architecture of the Anhinga Infrastructure

Figure 6 shows the architecture of the Anhinga Infrastructure. To distributed applications running on ad hoc networks of small mobile wireless devices, the infrastructure provides three capabilities:

- A *Java environment* consisting of the Anhinga Virtual Machine (AVM) and Anhinga Device Profile (ADP) API (class library).
- A *distributed services infrastructure*, Jini Mobile Edition (JiniME), which is a variation of Jini Network Technology designed for small mobile wireless devices.
- A *distributed communication and collaboration infrastructure*, Anhinga Spaces, which provides a tuple space [19] patterned after JavaSpaces [20] designed for small mobile wireless devices.

4.1. Anhinga Java Environment

The AVM implements the Java 2 Micro Edition Connected Limited Device Configuration (J2ME CLDC) specification [3] with extensions needed by the Anhinga Infrastructure. The J2ME CLDC was designed to provide a Java environment that runs in small devices with limited memories (128K to 512K). To fit within that space, the J2ME CLDC

omits floating point types, uses a simpler bytecode verification scheme than the Java 2 Standard Edition (J2SE), and replaces or omits many of the class libraries in the J2SE. In particular, the file and network related APIs in J2SE's `java.io` and `java.net` packages are replaced by the Generic Connection Framework [3]; and J2SE's user-defined classloader, reflection, object serialization, marshaled object, and remote method invocation (RMI) capabilities are omitted.

Like their full-fledged counterparts Jini and JavaSpaces, JiniME and Anhinga Spaces are based on *mobile code:* the ability to move a Java object, including both its state and the code for its class's methods, from one device to another device, and to execute the object's methods in the destination device. Unfortunately, the J2SE capabilities that enable mobile code are the very capabilities left out of the J2ME CLDC because of their large memory footprint. To support JiniME and Anhinga Spaces, the AVM has to add a mobile code capability back in. The challenge will be to provide a *lightweight* mobile code capability that is powerful and secure enough to build distributed applications while not occupying too large a memory footprint in small devices.

The AVM extensions to the J2ME CLDC specification include:

- *M2MP support.* A protocol stack for M2MP is layered on top of the wireless networking hardware provided by the device platform, such as wireless Ethernet (IEEE 802.11) or Bluetooth. The protocol stack consists of an M2MP message processing module and an adapter module for the particular networking hardware. In the case of Bluetooth, the adapter module uses the Bluetooth Host Controller Interface (HCI) to control the Bluetooth radio.
- *Dynamic classloading via M2MP*. As will be seen, JiniME requires the ability to download new Java classes from the network and install them in a running virtual machine. In the J2SE, this is accomplished by user-defined classloaders; however, the J2ME CLDC forbids user-defined classloaders for reasons of security. Accordingly, the AVM has the built-in ability to obtain Java classfiles from other devices in the ad hoc network as well as from its own device's local storage. If a classfile cannot be found locally, the requesting device broadcasts an M2MP message saying, "I need the classfile for class `com.foo.Bar`." In response, the device that has the classfile broadcasts an M2MP message saying, "Here is the classfile for class `com.foo.Bar`: . . ." Either individual classfiles or Java Archive (JAR) files containing multiple classfiles may be downloaded this way.
- *Security extensions.* To save space, the J2ME CLDC uses a simple "sandbox" security model rather than the highly flexible but heavyweight security model of J2SE. This simple security model will probably not be adequate when running downloaded code. Designing an adequate lightweight security model is an area for research. One area of investigation is *decentralized authentication techniques* which do not rely on a central authentication server or certificate authority, such as zero-knowledge proofs of identity [21], [22], [23]. Another area of investigation is *resource negotiation*, where a downloaded object is annotated with the resources it needs (security permissions, amount of memory, number of threads, and so on), the AVM checks whether it is willing to provide those resources, and if so the AVM ensures that the object stays within the agreed-upon limits.
- *Memory management.* The AVM ensures that a Java class is itself garbage collected when no instances of the class remain. This averts potential heap exhaustion due to buildup of no-longer-used code in a small device as objects and their classfiles are downloaded from the network.

The ADP, the Java class library for the Anhinga Java environment, is an extension of the Mobile Information Device Profile (MIDP) [4]. The MIDP defines APIs useful for programs running on small mobile devices, including user interface classes, classes for storing persistent data, and Generic Connection Framework classes for getting data from HTTP connections. The ADP adds the following APIs:

- *M2MP messaging.* Classes are provided to send

M2MP messages and to receive M2MP messages with specified message prefixes via the Generic Connection Framework.

- *Lightweight mobile code.* Classes are provided for lightweight reflection, object serialization, marshaled objects, and RMI. Remote method invocations are transported in M2MP messages under the hood. The caller broadcasts a message saying, "Will the object whose universally unique identifier is *X* please perform method *Y* with parameters *A, B,* and *C;* invocation identifier is *Z.*" After performing the method, the called object broadcasts a message saying, "Invocation *Z* returns normally with return value *R*" or "Invocation *Z* threw an exception *E.*"

4.2. Jini Mobile Edition

The Anhinga Infrastructure supports distributed services using the distributed systems paradigm espoused by Jini Network Technology [10], namely:

- Each device provides *services.*
- Each service is defined by a well-known *service interface.*
- A device allows others to use its services by providing *service proxy objects* which implement the service interfaces.
- A providing device's service proxy object is *moved into* the using device, which then uses the service by invoking the service proxy object's methods.
- The service proxy object *may implement the service interface however its designer wants* — for example, by communicating back to the providing device using a standard or specialized wire protocol, by executing solely within the using device, or by a hybrid of those approaches.
- There is a standard *discovery and lookup mechanism* whereby devices discover each other's appearance, discover each other's offered services, and obtain the corresponding service proxy objects.

A group of services that have published their existence, have discovered each other, and are interacting with each other is called a *federation.*

To enable the service proxy objects to execute in whatever devices they land in, regardless of the particular destination devices' operating environments, the service proxy objects must be compiled for a platform-independent target environment. Java is ideally suited for this purpose. For this reason, the Anhinga Infrastructure is based on Java.

Where Jini Network Technology is designed to use the capabilities of the J2SE (which is too heavyweight to run in small mobile devices), JiniME is designed to fit within the capabilities of the J2ME CLDC as augmented by the AVM and ADP. JiniME provides the following capabilities:

- *Self-hosted lookup services.* Each device provides its own JiniME Lookup Service, which lets others query the services provided on the device and download their service proxy objects. While in standard Jini each device would register its services with a separate Lookup Service, the Anhinga Infrastructure cannot rely on the availability of central servers. Instead, in JiniME each device provides its own Lookup Service for its own services.
- *Lookup service discovery via M2MP.* To discover the existence of other devices offering JiniME services, a discovery protocol is used. The requesting device broadcasts an M2MP message saying, "Please send your JiniME Lookup Service proxy object." The other devices comply. The requesting device can then call methods on the Lookup Service proxy objects obtained to query the services available on the other devices and download service proxy objects for those services.
- *Distributed programming infrastructure.* JiniME provides classes and interfaces for common distributed programming patterns, including leasing and remote events, analogous to those of Jini.
- *Sessile Bridge.* If a sessile host with access to both the wired and wireless networks is available, the host can run a "Sessile Bridge" application to provide JiniME services to a standard Jini federation and vice versa (Figure 7). The Sessile Bridge finds all the JiniME services on the wireless side, automatically constructs Jini service proxy objects that delegate to the corresponding JiniME service proxy objects, and reg-

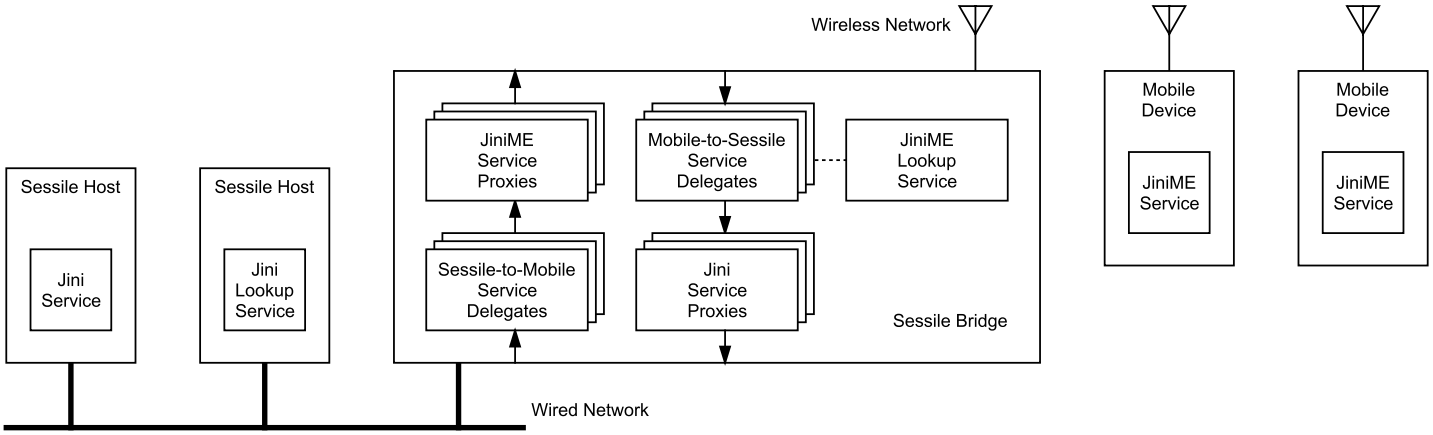


Figure 7. Sessile Bridge

isters the Jini service proxy objects with the Jini Lookup Service. The Sessile Bridge also finds all the Jini services on the wired side, automatically constructs JiniME service proxy objects that delegate to the corresponding Jini service proxy objects, and provides a JiniME Lookup Service with those services. When an application on one side of the boundary invokes a method on a service on the other side of the boundary, the invocation comes to the delegate running on the Sessile Bridge, which forwards the invocation on to the service.

4.3. Anhinga Spaces

The Anhinga Infrastructure supports distributed communication and collaboration using the "tuple space" abstraction for parallel and distributed programming originated by Gelernter [19] and embodied in such systems as Linda [24] and JavaSpaces [20]. Distributed applications use a tuple space in this fashion:

- There is a conceptual global *tuple space* shared by all the collaborating processes.
- The tuple space contains *tuples.* Each tuple is a typed data structure containing one or more information items. Each item in a tuple may be anything at all, from a simple integer to a full-blown object with state and behavior.
- Processes can *write* tuples into the tuple space.
- Processes can *read* tuples from the tuple space — the reading process gets a copy of the tuple, but the tuple stays in the tuple space for other processes to read or take.
- Processes can also *take* tuples out of the tuple space.
- To choose which tuple to read or take, the contents of the tuples in the tuple space are *matched* against the contents of a *template* supplied by the reading or taking process, and a tuple that matches the template is read or taken. If there are no matching tuples, the reading or taking process blocks until a matching tuple is written.

The writing and taking of tuples in the tuple space provides an interprocess communication and synchronization mechanism which is based on the contents of the tuples rather than on the identities of the processes. In other words, a process that writes a tuple doesn't need to know which process will take the tuple, and a process that takes a tuple doesn't need to know which process originally wrote the tuple. This capability is ideally suited for building a distributed system of small mobile wireless devices where the devices continually appear and disappear in an ad hoc fashion as the system operates.

Anhinga Spaces provides a lightweight tuple space implementation that does not rely on a central server to store the tuples. Rather, the tuples are distributed among the collaborating devices. Each tuple is initially stored in the device that wrote it, and each tuple is then moved into the device that takes it. The devices communicate among themselves to notify each other of tuples written and taken. If a

device goes down (or goes out of range of the other collaborating devices), tuples stored in that device will disappear from the tuple space, but the other devices' tuples will remain in the tuple space.

The principal challenge for a lightweight tuple space implementation will be to design a communication protocol for use in the wireless network that allows devices to access the tuple space in an ad hoc fashion and allows devices to write, read, and take tuples while not consuming too much network bandwidth. One possibility would be for a device writing a tuple to broadcast an M2MP message with a (short) hash of the tuple's contents, sufficient for another device to decide (with a high probability) whether to take the tuple. The writing device would send the complete tuple only if some other device indicates a tentative match based on the hash, whereupon the receiving device can examine the tuple's complete contents and decide whether to take the tuple. Multiple devices attempting to read and take the same tuples send M2MP messages to synchronize with each other and ensure the semantics of the tuple space operations are obeyed.

4.4. Relationship to the Jini Surrogate Architecture

Like the Anhinga Infrastructure, the Jini Surrogate Architecture [25], currently under development by the Jini Community, will also provide a way for non-J2SE-capable devices to participate in a Jini federation. Each device provides a *surrogate object*, a Java object. The Jini federation provides a *surrogate host* with copious memory, a full J2SE environment, and a network connection to the Jini federation. Devices upload their surrogate objects to the surrogate host, which executes the devices' surrogate objects. Each surrogate object exports its device's services to the Jini federation and finds and uses other services on the device's behalf, communicating back to the device using a private protocol. The Jini Surrogate Architecture provides for a variety of interconnection mechanisms between the devices and the surrogate host, including wireless connections [26], thus allowing small mobile wireless devices to make use of surrogate hosts.

However, since the Jini Surrogate Architecture is a client-server architecture requiring a sessile server (the surrogate host), it suffers from the drawbacks discussed in Section 1.2 when contemplated as the sole basis for a distributed services infrastructure for small mobile wireless devices. The Anhinga Infrastructure provides an alternative to the Jini Surrogate Architecture that does not require the presence of a sessile surrogate host, thus allowing small mobile wireless devices to engage in distributed applications among themselves in places where there is no wireless access to the wired network.

5. Status and Future Plans

The Anhinga Infrastructure is a work in progress. The initial architecture has been designed and exploratory implementation is underway. Preliminary investigations have produced a prototype lightweight RMI library and self-hosted Lookup Service [27] and a prototype M2MP protocol stack written in Java; the prototypes are available from the Anhinga Project web site [11].

Research plans for the Many-to-Many Protocol include implementing M2MP on IEEE 802.11 and Bluetooth physical layers, investigating the practicality of implementing a broadcast medium on top of Bluetooth, investigating the transmission error and packet loss characteristics of wireless interconnections among proximal devices to determine whether it is a valid assumption that the link is mostly reliable, and investigating packet flooding algorithms for relaying messages throughout geographically extended groups of devices.

Research plans for the Anhinga Infrastructure include implementing the AVM and ADP on PCs and PDAs running Linux, investigating how to implement lightweight mobile code capabilities including security, designing broadcast protocols for a lightweight distributed tuple space, investigating the potential for synergy between the Anhinga Infrastructure and the Jini Surrogate Architecture, and developing proof-of-concept applications such as those mentioned in Section 2.

6. Acknowledgments

The Anhinga Project is funded by a grant from Sun Microsystems.

I would like to thank Hans-Peter Bischof, Jona-

than Coles, Steve Hanna, Brian Koponen, Jeffrey Lasky, Jeffrey Myers, Jacob Rigby, and Jim Waldo, for their many conversations and helpful ideas.

7. References

[1] Bluetooth Special Interest Group. *Specification of the Bluetooth System, Version 1.1.* February 22, 2001. `http://www.bluetooth.com/developer/specification/specification.asp`

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition.* Reading, MA: Addison-Wesley, 2000.

[3] Sun Microsystems. *Java 2 Platform Micro Edition Connected, Limited Device Configuration Specification Version 1.0.* May 19, 2000. `http://java.sun.com/aboutJava/communityprocess/final/jsr030/index.html`

[4] Sun Microsystems. *Mobile Information Device Profile (JSR-37) JCP Specification, Java 2 Platform, Micro Edition, 1.0.* September 1, 2000. `http://java.sun.com/aboutJava/communityprocess/final/jsr037/index.html`

[5] Sun Microsystems. *PDA Profile for J2ME* (in progress). `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_075_pda.html`

[6] Sun Microsystems. *Java APIs for Bluetooth* (in progress). `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_082_bluetooth.html`

[7] Internet Engineering Task Force. IP Routing for Wireless/Mobile Hosts (MobileIP) Working Group. `http://www.ietf.org/html.charters/mobileip-charter.html`

[8] Internet Engineering Task Force. Mobile Ad Hoc Networks (MANET) Working Group. `http://www.ietf.org/html.charters/manet-charter.html`

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.4.1.* November 2000.

[10] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Reading, MA: Addison-Wesley, 1999.

[11] The Anhinga Project. `http://www.cs.rit.edu/~anhinga`

[12] Charles E. Perkins and Pravin Bhagwat. "DSDV Routing Over a Multihop Wireless Network of Mobile Computers." In Tomasz Imielinski and Henry F. Korth, editors, *Mobile Computing* (Boston, MA: Kluwer Academic Publishers, 1996), pages 183–206.

[13] David B. Johnson, David A Maltz, and Josh Broch. "DSR: The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks." In Charles E. Perkins, editor, *Ad Hoc Networking* (Boston, MA: Addison-Wesley, 2001), pages 139–172.

[14] Charles E. Perkins and Elizabeth M. Royer. "The Ad Hoc On-Demand Distance-Vector Protocol." In Charles E. Perkins, editor, *Ad Hoc Networking* (Boston, MA: Addison-Wesley, 2001), pages 173–219.

[15] Zygmunt J. Haas and Marc R. Pearlman. "ZRP: A Hybrid Framework for Routing in Ad Hoc Networks." In Charles E. Perkins, editor, *Ad Hoc Networking* (Boston, MA: Addison-Wesley, 2001), pages 221–253.

[16] J. J. Garcia-Luna-Aceves and Marcelo Spohn. "Bandwidth-Efficient Link-State Routing in Wireless Networks." In Charles E. Perkins, editor, *Ad Hoc Networking* (Boston, MA: Addison-Wesley, 2001), pages 323–350.

[17] David R. Cheriton and Dale Skeen. "Understanding the limitations of causally and totally ordered communication." *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles,* December 5–8, 1993, Asheville, NC, pages 44–57.

[18] D. Meyer. "Administratively Scoped IP Multicast." Internet Request For Comments (RFC) 2365, July 1998.

[19] David Gelernter. “Generative Communication in Linda.” *ACM Transactions on Programming Languages and Systems,* Volume 7, Number 1, January 1985, pages 80–112.

[20] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice.* Reading, MA: Addison-Wesley, 1999.

[21] U. Feige, A. Fiat, and A. Shamir. “Zero Knowledge Proofs of Identity.” *Journal of Cryptology,* Volume 1, Number 2, 1988, pages 77–94.

[22] L. Guillou and J. Quisquater. “A Practical Zero-Knowledge Protocol Fitted to Security Microprocessor Minimizing Both Transmission and Memory.” *Advances in Cryptology — EUROCRYPT ’88 Workshop on the Theory and Application of Cryptographic Techniques,* Davos, Switzerland, May 25–27, 1988, pages 123–128.

[23] C. Schnorr. “Efficient Signature Generation for Smart Cards.” *Journal of Cryptology,* Volume 4, Number 3, 1991, pages 161–174.

[24] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course.* Cambridge, MA: MIT Press, 1990.

[25] Jini Surrogate Project. `http://developer.jini.org/exchange/projects/surrogate/`

[26] Jini Wireless Device Project. `http://developer.jini.org/exchange/projects/wirelessdevice/`

[27] Alan Kaminsky. “Running Jini Network Technology in Small Places.” *Fifth Jini Community Meeting,* Amsterdam, the Netherlands, December 12, 2000. `http://www.jini.org/jini5/slides/Small_Places/index.htm`