Fundamentals of Mobile and Pervasive Computing

Frank Adelstein
Sandeep K. S. Gupta
Golden G. Richard III
Loren Schwiebert

McGraw-Hill
New York Chicago San Francisco Lisbon London Madrid
Mexico City Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

The McGraw-Hill Companies

CIP Data is on file with the Library of Congress

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 1 0 9 8 7 6 5 4

ISBN 0-07-141237-9

The sponsoring editor for this book was Stephen S. Chapman and the production supervisor was Sherri Souffrance. It was set in Century Schoolbook by International Typesetting and Composition. The art director for the cover was Anthony Landi.

Printed and bound by RR Donnelley.

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, McGraw-Hill Professional, Two Penn Plaza, New York, NY 10121-2298. Or contact your local bookstore.

This book is printed on recycled, acid-free paper containing a minimum of 50% recycled, de-inked fiber.

Chapter

6

Middleware for Application Development: Adaptation and Agents

Application development for mobile computers is a difficult task—on their own, applications are faced with a myriad of challenges: limited power and processing speed, varying levels of network connectivity, completely disconnected operation, and discovery of needed services. The goal of mobile middleware is to provide abstractions that reduce development effort, to offer programming paradigms that make developing powerful mobile applications easier, and to foster interoperability between applications. *Service discovery*, or the art of dynamically discovering and advertising services, is the subject of the next chapter. This chapter examines two other important types of middleware for mobile computing—*adaptation* and *agents*. The first, adaptation, was first discussed in Chap. 1. We revisit this topic in this chapter. Recall that adaptation helps applications to deal intelligently with limited or fluctuating resource levels. The second type of middleware, mobile agents, provides a powerful and flexible paradigm for access to remote data and services.

6.1 Adaptation

Mobile computers must execute user- and system-level applications subject to a variety of resource constraints that generally can be ignored in modern desktop environments. The most important of these constraints are power, volatile and nonvolatile memory, and network bandwidth, although other physical limitations such as screen resolution

are also important. In order to provide users with a reasonable computing environment, which approaches the best that currently available resources will allow, applications and/or system software must adapt to limited or fluctuating resource levels. For example, given a sudden severe constraint on available bandwidth, a mobile audio application might stop delivering a high-bit-rate audio stream and substitute a lower-quality stream. The user is likely to object less to the lower-quality delivery than to the significant dropouts and stuttering if the application attempted to continue delivering the high-quality stream. Similarly, a video application might adjust dynamically to fluctuations in bandwidth, switching from high-quality, high-frame-rate color video to black-and-white video to color still images to black-and-white still images as appropriate. A third example is a mobile videogame application adjusting to decreased battery levels by modifying resolution or disabling three-dimensional (3D) features to conserve power.

6.1.1 The spectrum of adaptation

At one end of the spectrum, adaptation may be entirely the responsibility of the mobile computer's operating system (OS); that is, the software for handling adaptation essentially is tucked under the OS hood, invisible to applications. At the other end, adaptation may be entirely the responsibility of individual applications; that is, each application must address all the issues of detecting and dealing with varying resource levels. Between these extremes, a number of *application-aware strategies* are possible, where the OS and individual application each share some of the burden of adaptation. While applications are involved in adaptation decisions, the middleware and/or OS provides support for resource monitoring and other low-level adaptation functions. The spectrum of adaptation is depicted in Fig. 6.1. In this part of the chapter, we are concerned primarily with middleware for adaptation, that is, software interfaces that allow applications to take part in the adaptation process. Pure system-level adaptation strategies, those which take place in a mobile-aware file system such as Coda (e.g., caching and hoarding), are covered elsewhere in this book.

6.1.2 Resource monitoring

All adaptation strategies must measure available resources so that adaptation policies can be carried out. For some types of resources—cash, for example—monitoring is not so difficult. The user simply sets limits and appropriate accounts. For others, more elaborate approaches are required. The Advanced Configuration and Power Interface (ACPI) provides developers with a standardized interface to power-level information on modern devices equipped with "smart" batteries. Accurately

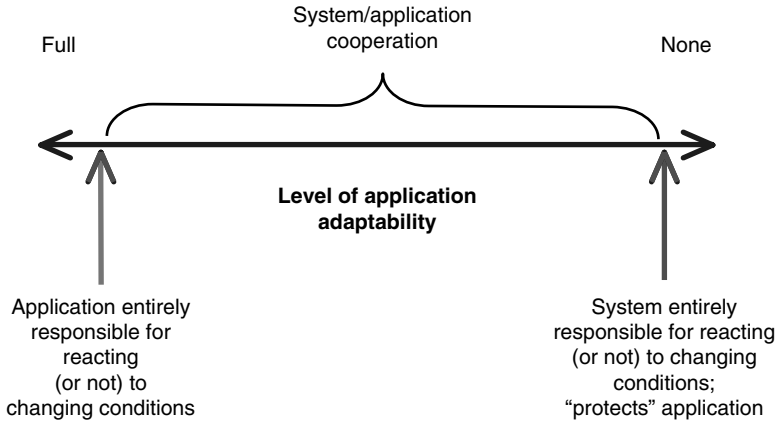

Figure 6.1 At one end of the spectrum of adaptation, applications are entirely responsible for reacting to changing resource levels. At the other end of the spectrum, the operating system reacts to changing resource levels without the interaction of individual applications.

measuring network bandwidth over multihop networks is more difficult. Some approaches are described in Lai and Baker (1999) for the interested reader. Whatever methods are used to measure resource levels have a direct impact on the effectiveness of the entire adaptation process because accurate measurement of resource levels is critical to making proper adaptation decisions.

6.1.3 Characterizing adaptation strategies

The Odyssey project (Noble et al., 1997; Noble, 2000) at Carnegie Mellon University was one of the first application-aware middleware systems, and it serves as a good model for understanding application-aware adaptation. In describing the Odyssey system, Satyanarayanan proposed several measures that are useful for classifying the goodness of an adaptation strategy. We describe these—*fidelity, agility*, and *concurrency*—below.

Fidelity measures the degree to which a data item available to an application matches a reference copy. The reference copy for a data item is considered the exemplar, the ideal for that data item—essentially, the version of the data that a mobile computer would prefer given no resource constraints. Fidelity spans many dimensions, including perceived quality and consistency. For example, a server might store a 30-frame-per-second (fps), 24-bit color depth video at 1600×1200 resolution in its original form as shot by a digital video camera. This reference copy of the video is considered to have 100 percent fidelity. Owing to resource constraints such as limited network bandwidth, a mobile host may have

to settle for a version of this video that is substantially reduced in quality (assigned a lower fidelity measure, perhaps 50 percent) or even for a sequence of individual black-and-white still frames (with a fidelity measure of 1 percent). If the video file on the server is replaced periodically with a newer version and a mobile host experiences complete disconnection, then an older, cached version of the video may be supplied to an application by adaptation middleware. Even if this cached version is of the same visual quality as the current, up-to-date copy, its fidelity may be considered lower because it is not the most recent copy (i.e., it is *stale*).

While some data-dependent dimensions of fidelity, such as the frame rate of a video or the recording quality of audio, are easily characterized, others, such as the extent to which a database table is out of date or a video is not the most current version available, do not map easily to a 0 to 100 percent fidelity scale. In cases where there is no obvious mapping, a user's needs must be taken into account carefully when assigning fidelity levels. More problematic is the fact that fidelity levels are in general type-dependent—there are as many different types of fidelity-related adaptations as there are types of data streams; for example, image compression schemes are quite different from audio compression schemes. Generally, an adaptation strategy should provide the highest fidelity possible given current and projected resource levels. Current adaptation middleware tends to concentrate on the present. Factoring projected resource levels into the equation is an area for future research.

Agility measures an adaptation middleware's responsiveness to changes in resource levels. For example, a highly agile system will determine quickly and accurately that network bandwidth has increased substantially or that a fresh battery has been inserted. *An adaptation middleware's agility directly limits the range of fidelity levels that can be accommodated.* This is best illustrated with several examples, which show the importance of both speed and accuracy. For example, if the middleware is very slow to respond to a large increase in network bandwidth over a moderate time frame (perhaps induced by a user resting in an area with 802.11 WLAN connectivity), then chances to perform opportunistic caching, where a large amount of data are transferred and hoarded in response to high bandwidth, may be lost. Similarly, an adaptation middleware should notice that power levels have dropped substantially before critical levels are reached. Otherwise, a user enjoying a high-quality (and power-expensive) audio stream may be left with nothing, rather than a lower-quality audio stream that is sustainable.

Agility, however, is not simply a measure of the speed with which resource levels are measured; accuracy is also extremely important. For example, consider an 802.11a wireless network, which is much more sensitive to line-of-sight issues than 802.11b or 802.11g networks. A

momentary upward spike in available bandwidth, caused by a mobile host connected to an 802.11a network momentarily having perfect line of sight with an access point, should not necessarily result in adjustments to fidelity level. If such highly transient bandwidth increases result in a substantial increase in fidelity level of a streaming video, for example, many frames may be dropped when bandwidth suddenly returns to a lower level.

The last measure for adaptation middleware that we will discuss is *concurrency*. Although the last generation of PDAs (such as the original Pilot by Palm, Inc.) used single-threaded operating systems, capable of executing only one application at a time; newer PDAs, running newer versions of Palm OS, variants of Microsoft Windows, and Linux, run full-featured multitasking OSs. Thus it is reasonable to expect that even the least powerful of mobile devices, not to mention laptops that run desktop operating systems, will execute many concurrent applications, all of which compete for limited resources such as power and network bandwidth. This expectation has a very important implication for adaptation: Handling adaptation at the left end of the spectrum (as depicted in Fig. 6.1), where individual applications assume full responsibility for adapting to resource levels, is probably not a good idea. To make intelligent decisions, each application would need to monitor available resources, be aware of the resource requirements of all other applications, and know about the adaptation decisions being made by the other applications. Thus some system-level support for resource monitoring, where the OS can maintain the "big picture" about available resources needs and resource levels, is important.

6.1.4 An application-aware adaptation architecture: Odyssey

In this section we examine the Odyssey architecture in greater detail. In the spectrum of adaptation, Odyssey sits in the middle—applications are *assisted* by the Odyssey middleware in making decisions concerning fidelity levels. Odyssey provides a good model for understanding the issues in application-aware adaptation because the high-level architecture is clean, and the components for supporting adaptation are clearly delineated. The Odyssey architecture consists of several high-level components: the *interceptor*, which resides in the OS kernel, the *viceroy,* and one or more *wardens*. These are depicted in Fig. 6.2. The version of Odyssey described in Nobel and colleagues (1997) runs under NetBSD; more recent versions also support Linux and FreeBSD. To minimize changes to the OS kernel, Odyssey is implemented using the Virtual File System (VFS) interface, which is described in great detail for kernel hacker types in Bovet and Cesati (2002). Applications interact with

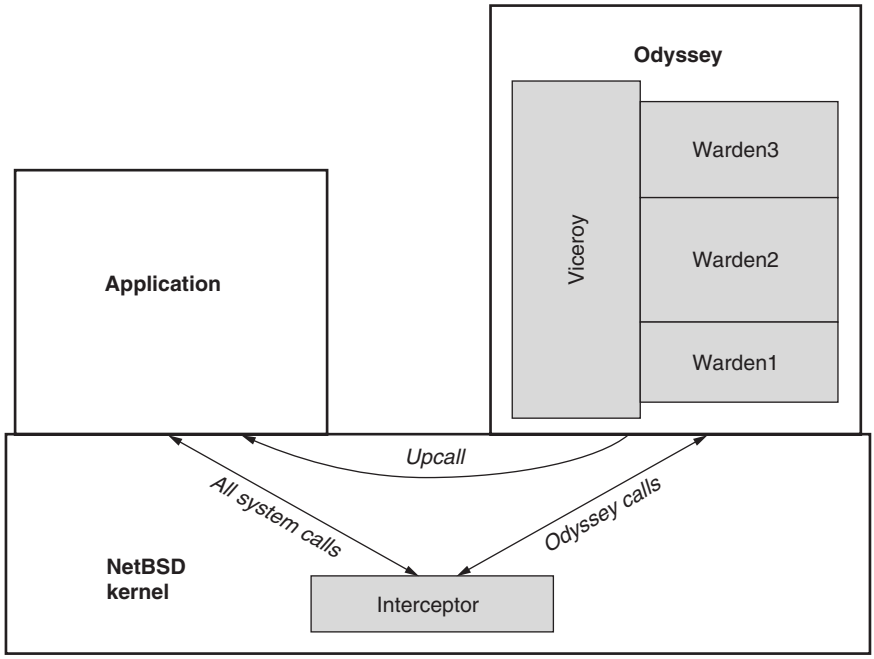

Figure 6.2 The Odyssey architecture consists of a type-independent viceroy and a number of type-specific wardens. Applications register windows of acceptable resource levels for particular types of data streams and receive notifications is when current resource levels fall outside the windows.

Odyssey using (mostly) file system calls, and the interceptor, which resides in the kernel, performs redirection of Odyssey-specific system calls to the other Odyssey components.

The basic Odyssey model is for an application to choose a fidelity level for each data type that will be delivered—e.g., 320×240 color video at 15 fps. The application then computes resource needs for delivery of each stream and registers these needs with Odyssey in the form of a "window" specifying minimum and maximum need. The viceroy monitors available resources and generates a callback to the application when available resources fall outside registered resource-level window. The application then chooses a new fidelity level, computes resource needs, and registers these needs, as before. Thus applications are responsible for deciding fidelity levels *and* computing resource requirements—the primary contribution that Odyssey makes is to monitor resources and to notify applications when available resources fall outside constraints set by the application. Before describing a sample Odyssey application, the wardens and viceroy are discussed in detail below.

Wardens. A *warden* is a type-specific component responsible for handling all adaptation-related operations for a particular sort of data stream (e.g., a source of digital images, audio, or video). Wardens sit between an application and a data source, handling caching and arranging for delivery of data of appropriate fidelity levels to the application. A warden must be written for each type of data source. An application typically must be partially rewritten (or an appropriate proxy installed) to accept data through a warden rather than through a direct connection to a data source, such as a streaming video server.

Viceroy. In Odyssey, the viceroy is a type-independent component that is responsible for global resource control. All the wardens are statically compiled with the viceroy. The viceroy monitors resource levels (e.g., available network bandwidth) and initiates callbacks to an application when current resource levels fall outside a range registered by the application. The types of resources to be monitored by the viceroy in Odyssey include network bandwidth, cache, battery power, and CPU, although the initial implementations of the Odyssey architecture did not support all these resource types.

6.1.5 A sample Odyssey application

We now turn to one of the sample applications discussed in Nobel and colleagues (1997): the *xanim* video player. The xanim video player was modified to use Odyssey to adapt to varying network conditions, with three fidelity levels available—two levels of JPEG compression and black-and-white frames. The JPEG compression frames are labeled 99 and 50 percent fidelity, whereas the black-and-white content is labeled 1 percent fidelity. Integration of xanim with Odyssey is illustrated in Fig. 6.3. A "video warden" prefetches frames from a video server with the appropriate fidelity and supplies the application with metadata for the video being played and with individual frames of the video.

The performance of the modified xanim application was tested using simulated bandwidths of 140 kB/s for "high" bandwidth and 40 kB/s for "low" bandwidth. A number of strategies were used to vary bandwidth: *step up*, which holds bandwidth at the low level for 30 seconds, followed by an abrupt increase to high bandwidth for 30 seconds; *step down*, which reverses the bandwidth levels of step up but maintains the same time periods; *impulse up*, which maintains a low bandwidth over a 60-second period with a single 2-second spike of high bandwidth in the middle; and *impulse down*, which maintains high bandwidth for 60 seconds with a single 2-second spike of low bandwidth in the middle. Both high and low bandwidth levels are able to support black-and-white video and the lower-quality (50 percent fidelity) JPEG video. Only the high bandwidth level

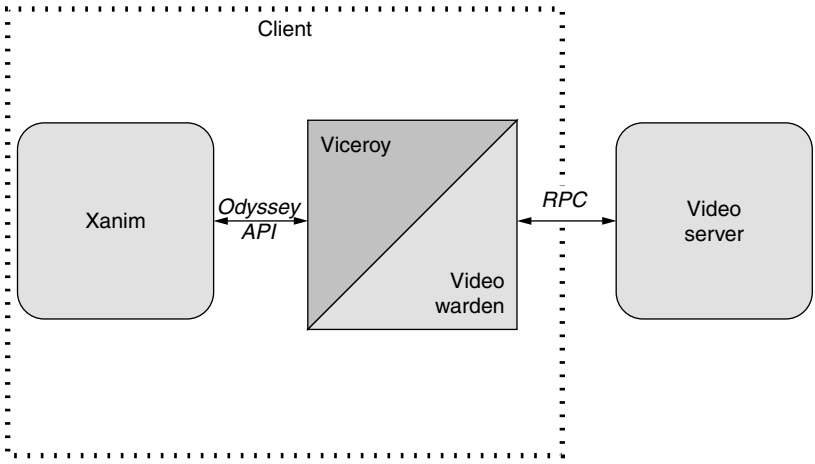

Figure 6.3 Architecture of the adapted video player in .y.

is sufficient for the 99 percent fidelity JPEG frames to be delivered without substantial numbers of dropped frames.

In the tests, Odyssey maintained average fidelities of 73, 76, 50, and 98 percent for step up, step down, impulse up, and impulse down, respectively, all with less than 5 percent dropped frames. In contrast, trying to maintain the 99 percent fidelity rate by transferring high-quality video at all times, ignoring available network bandwidth, resulted in losses of 28 percent of the frames for step up and step down and 58 percent of the frames for impulse up. Several other adapted applications are discussed in the Odyssey publications.

6.1.6 More adaptation middleware

Puppeteer. For applications with well-defined, published interfaces, it is possible to provide adaptation support without modifying the applications directly. The Puppeteer architecture allows component-based applications with published interfaces to be adapted to environments with poor network bandwidth (a typical situation for mobile hosts) without modifying the application (de Lara, Wallach, and Zwaenepoel, 2001). This is accomplished by outfitting applications and data servers with custom proxies that support the adaptation process. A typical application adaptation under Puppeteer is a retrofit of Microsoft PowerPoint to support incremental loading of slides from a large presentation or support for progressive JPEG format to speed image loading. Both these adaptations presumably would enhance a user's experience when handling a large PowerPoint presentation over a slow network link.

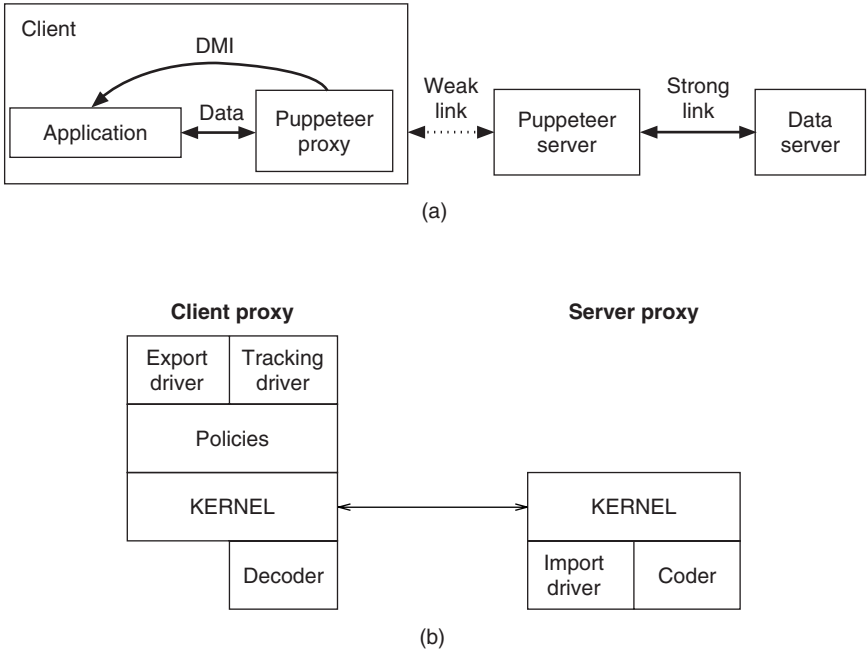

Figure 6.4 (a) Illustrates the overall Puppeteer architecture, where client applications interact with data servers through proxies. DMI is the Data Manipulation Interface of the applications, which allows Puppeteer to view and modify data acted on by the application. The relationship between client-side and server-side proxies is illustrated in (b).

The Puppeteer architecture is depicted in Fig. 6.4. The Puppeteer provides a kernel that executes on both the client and server side proxies, supporting a document type called the Puppeteer Intermediate Format (PIF), a hierarchical, format-neutral format. The kernel also handles all communication between client and server sides. To adapt a document, the server and client side proxies communicate to establish a high-level PIF skeleton of the document. Adaptation policies control which portions of the document will be transferred and which fidelities will be chosen for the transmitted portions. For example, for a Microsoft PowerPoint document, selected slides may transferred, with images rendered at a lower fidelity than in the original presentation. The *import driver* and *export driver* parse native document format to PIF and PIF to native document format, respectively. Transcoders in Puppeteer perform transformations on data items to support the adaptation policies. For example, a Puppeteer transcoder may reduce the quality of JPEG images or support downloading only a subset of a document's data. A typical Puppeteer-adapted application operates as follows:

- When the user opens a document, the Puppeteer kernel instantiates an appropriate import driver on the server side.
- The import driver parses the native document format and creates a PIF format document. The skeleton of the PIF is transmitted by the kernel to the client-side proxy.
- On the client side, policies available to the client-side proxy result in requests to transfer selected portions of the PIF (at selected fidelities) from the server side. These items are rendered by the export driver into native format and supplied to the application through its well-known interface.
- At this point, the user regains control of the application. If specified by the policy, additional portions of the requested document can be transferred by Puppeteer in the background and supplied to the application as they arrive.

Coordinating adaptation for multiple mobile applications. Efstatiou and colleagues (2002a, 2002b) argue that what's missing from most current adaptation middleware architectures is coordination among adaptive applications. Odyssey and Puppeteer, for example, support sets of independently adapting applications but do not currently assist multiple applications in coordinating their adaptation strategies. When multiple applications are competing for shared resources, individual applications may make decisions that are suboptimal. At least three issues are introduced when multiple applications attempt to adapt to limited resources—*conflicting adaptation, suboptimal system operation*, and *suboptimal user experience*.

Several sample scenarios illustrate these concerns. First, consider a situation where a number of applications executing on a mobile host with limited power periodically write data to disk. This would occur, for example, if two or more applications with automatic backup features were executing. Imagine that the mobile host maintains a powered-down state for its hard drives to conserve energy. Then, each time one of the automatic backup facilities executes, a hard disk on the system must be spun up. If the various applications perform automatic backups at uncoordinated times, then the disk likely will spin up quite frequently, wasting a significant amount of energy. If the applications coordinated to perform automatic backups, on the other hand, then disk writes could be performed "in bulk," maximizing the amount of time that the disk could remain powered down. This example illustrates suboptimal system operation despite adaptation.

Another issue when multiple applications adapt independently is conflicting adaptation. Imagine that one application is adapting to varying power, whereas another application is adapting to varying network

bandwidth. When the battery level in the mobile device becomes a concern, then the power-conscious application might throttle its use of the network interface. This, in turn, makes more bandwidth available, which might trigger the bandwidth-conscious application to raise fidelity levels for a data stream, defeating the other application's attempt to save energy.

A third issue is that in the face of limited resources, a user's needs can be exceedingly difficult to predict. Thus some user participation in the adaptation process probably is necessary. To see this, imagine that a user is enjoying a high-bandwidth audio stream (Miles Davis, *Kind of Blue*?) while downloading a presentation she needs to review in 1 hour. With abundant bandwidth, both applications can be well served. However, if available bandwidth decreases sharply (because an 802.11 access point has gone down, for example, and the mobile host has fallen back to a 3G connection), should a lower-quality stream be chosen and the presentation download delayed because Miles is chewing up a few tens of kilobits per second? Or should the fun stop completely and the work take precedence?

Efstatiou and colleagues propose using an adaptation policy language based on the event calculus (Kowalsky, 1986) to specify global adaptation policies. The requirements for their architecture are that a set of extensible adaptation attributes be sharable among applications, that the architecture be able to centrally control adaptation behavior, and that flexible, system-wide adaptation policies, depending on a variety of issues, be expressible in a policy language. Their architecture also allows human interaction in the adaptation process both to provide feedback to the user and to engage the user in resolving conflicts (e.g., Miles Davis meets downloading PowerPoint). Applications are required to register with the system, providing a set of adaptation policies and modes of adaptation supported by the application. In addition, the application must expose a set of state variables that define the current state of the application. Each application generates events when its state variables change in meaningful ways so that the adaptation architecture can determine if adaptive actions need to be taken; for example, when a certain application is minimized, a global adaptation policy may cause that application to minimize its use of system resources. A *registry* in the architecture stores information about each application, and an *adaptation controller* monitors the state of the system, determining when adaptation is necessary and which applications should adapt. Another policy language–driven architecture advocating user involvement is described in Keeney and Cahill (2003).

6.2 Mobile Agents

We now turn to another type of mobile middleware, mobile agent systems. Almost all computer users have used mobile code, whether they realize

it or not—modern browsers support Javascript, Java applets, and other executable content, and simply viewing Web pages results in execution of the associated mobile code. Applets and their brethren are mostly *static*, in that code travels from one or more servers to a client and is executed on the client. For security reasons, the mobile code often is prevented from touching nonlocal resources. Mobile agents are a significant step forward in sophistication, supporting the migration of not only code but also state. Unlike applets, whose code typically travels (at an abstract level at least) one "hop" from server to client, mobile agents move freely about a network, making autonomous decisions on where to travel next. Mobile agents have a mission and move about the network extracting data and communicating with other agents in order to meet the mission goals.

Like adaptation middleware, mobile agent systems (e.g., Cabri, Leonardi, and Zambonelli, 2000; Gray, 1996, 1997; Gray et al., 1998, 2000; Bradshaw et al., 1999; Lange and Oshima, 1998; Peine and Stoplmann, 1997; Wong, Paciorek, and Moore, 1999; Wong et al., 1997) support execution of mobile applications in resource-limited environments, but mobile agent systems go far beyond allowing local applications to respond to fluctuating resource levels. A mobile agent system is a dynamic client-server (CS) architecture that supports the migration of mobile applications (agents) to and from remote servers. An agent can migrate whenever it chooses either because it has accomplished its task completely or because it needs to travel to another location to obtain additional data. An alternative to migration that an agent might exercise is to create one or more new agents dynamically and allow these to migrate. The main idea behind mobile agents is to get mobile code as close to the action as possible—mobile agents migrate to remote machines to perform computations and then return home with the goods.

For example, if a mobile user needs to search a set of databases, a traditional CS approach may perform remote procedure calls against the database servers. On the other hand, a mobile agents approach would dispatch one or more applications (agents) either directly to the database servers or to machines close to the servers. The agents then perform queries against the database servers, sifting the results to formulate a suitable solution to the mobile user's problem. Finally, the mobile agents return home and deliver the results.

The advantages of this approach are obvious. First, if bandwidth available to the mobile user is limited and the database queries are complicated, then performing a series of remote queries against the servers might be prohibitively expensive. Since the agents can execute a number of queries much closer to the database servers in order to extract the desired information, a substantial amount of bandwidth might be saved (of course, transmission of the agent code must be taken into account). Second, continuous network connectivity is not required.

The mobile user might connect to the network, dispatch the agent, and then disconnect. When the mobile user connects to the network again later, the agent is able to return home and present its results. Finally, the agents are not only closer to the action, but they also can be executed on much more powerful computers, potentially speeding up the mining of the desired information.

Of course, there are substantial difficulties in designing and implementing mobile agent systems. After briefly discussing the motivations for mobile agent systems in the next section, those challenges will consume the rest of this chapter.

6.2.1 Why mobile agents? And why not?

We first discuss the advantages of mobile agents at a conversational level, and then we look at the technical advantages and disadvantages in detail. First, a wide variety of applications can be supported by mobile agent systems, covering electronic commerce (sending an agent shopping), network resource management (an agent might traverse the network, checking versions of installed applications and initiating upgrades where necessary), and information retrieval (an agent might be dispatched to learn everything it can about Thelonious Monk).

An interesting observation made by Gray and colleagues (2000) is worth keeping in mind when thinking about agent-based applications: While *particular* applications may not make a strong case for deployment of mobile agent technology, *sets* of applications may make such a case. To see this point, consider the database query example discussed in the preceding section. Rather than using mobile agents, a custom application could be deployed (statically) on the database servers. This application accepts jobs (expressing the type of information required) from a mobile user, performs a sequence of appropriate queries, and then returns the results. Since most of the processing is done off the mobile host, the resource savings would be comparable to a mobile agents solution.

Similarly, little computational power on the mobile host is required because much of the processing can be offloaded onto the machine hosting the custom application. However, what if a slightly different application is desired by a mobile user? Then the server configuration must be changed. Like service discovery protocols, covered in Chap. 7, mobile agent systems foster creation of powerful, *personalized* mobile applications based on common frameworks. While individual mobile applications can be written entirely without the use of agent technologies, the amount of effort to support a changing *set* of customized applications may be substantially higher than if mobile agents were used.

Mobile agent systems provide the following set of technical advantages (Milojicic, Douglis, and Wheel, 1998):

- *The limitations of a single client computer are reduced.* Rather than being constrained by resource limitations such as local processor power, storage space, and particularly network bandwidth, applications can send agents "into the field" to gather data and perform computations using the resources of larger, well-connected servers.
- *The ability to customize applications easily is greatly improved.* Unlike traditional CS applications, servers in an agent system merely provide an execution *environment* for agents rather than running customized server applications. Agents can be freely customized (within the bounds of security restrictions imposed by servers) as the user's needs evolve.
- *Flexible, disconnected operation is supported.* Once dispatched, a mobile agent is largely independent of its "home" computer. It can perform tasks in the field and return home when connectivity to the home computer is restored. Survivability is enhanced in this way, especially when the home computer is a resource-constrained device such as a PDA. With a traditional CS architecture, loss of power on a PDA might result in an abnormal termination of a user's application.

Despite these advantages, mobile agent architectures have several significant disadvantages or, if that is too strong a word, disincentives. One is that neither a killer application nor a pressing need to deploy mobile agent technology has been identified. Despite their sexiness, mobile agents do not provide solutions to problems that are otherwise unsolvable; rather, they simply seem to provide a good framework in which to solve certain problems. In reflections on the Tacoma project (Milojicic, Douglis, and Wheel, 1998), Johansen, Schneider, and van Renesse note that while agents potentially reduce bandwidth and tolerate intermittent communication well, bandwidth is becoming ever more plentiful, and communication is becoming more reliable. As wireless networking improves and mobile devices become more powerful and more prevalent, will mobile agents technologies become less relevant? Further, while a number of systems exist, they are largely living in research laboratories. For mobile agent systems to meet even some of their potential, widespread deployment of agent environments is required so that agents may travel freely about the Internet.

A related problem is a lack of standardization. Most mobile agent systems are not interoperable. Some effort has gone into interoperability for agent systems, but currently, there seem to be no substantial market pressures forcing the formation of a single (or even several) standards for mobile agent systems. The Mobile Agents System Interoperability Facility (MASIF; Milojicic et al., 1999) is one early attempt at fostering agent interoperability for Java-based agent systems.

All the disadvantages just discussed are surmountable with a little technical effort—apply a good dose of marketing, and most disappear. There is a killer disadvantage, however, and that is *security*. Even applets and client-side scripting languages (such as Javascript), which make only a single hop, scare security-conscious users to death, and many users turn off Java, Javascript, and related technologies in their Web browsers. Such users maintain this security-conscious stance even when interacting with Web sites in which they place significant trust because the potential for serious damage is high should the sandbox leak. Security for mobile agent systems is far more problematic than simple mobile code systems such as Java applets because agents move autonomously.

There are at least two broad areas of concern. First, agents must be prevented from performing either unintentional or malicious damage as they travel about the network. Could an agent have been tampered with at its previous stop? Is it carrying a malicious logic payload? Does it contain contraband that might be deposited on a machine? Will the agent use local resources to launch a denial-of-service attack against another machine? Essentially, if agents are to be allowed to get "close to the action," then the "action" must be convinced (and *not* just with some marketing) that the agents will not destroy important data or abuse resources. Second, the agents themselves must be protected from tampering by malicious servers. For example, an agent carrying credit card information to make purchases on behalf of its owner should be able to control access to the credit card number. Similarly, an agent equipped with a proprietary data-mining algorithm should be able to resist reverse engineering attacks as it traverses the network.

6.2.2 Agent architectures

To illustrate the basic components of mobile agent architectures, a high-level view of Telescript (White in Milojicic, Douglic, and Wheel, 1998) works well. Telescript was one of the first mobile agent systems, and while it is no longer under development, many subsequent systems borrowed ideas from Telescript. There are a number of important components in the Telescript architecture: *agents, places, travel, meetings, connections, authorities*, and *permits*. These are depicted in Fig. 6.5. Each of these components is described in detail below.

Places. In a mobile agent system, a network is composed of a set of places—each place is a location in the network where agents may visit. Each place is hosted by a server (or perhaps a user's personal computer) and provides appropriate infrastructure to support a mobile agent migrating to and from that location. Servers in a network that do not

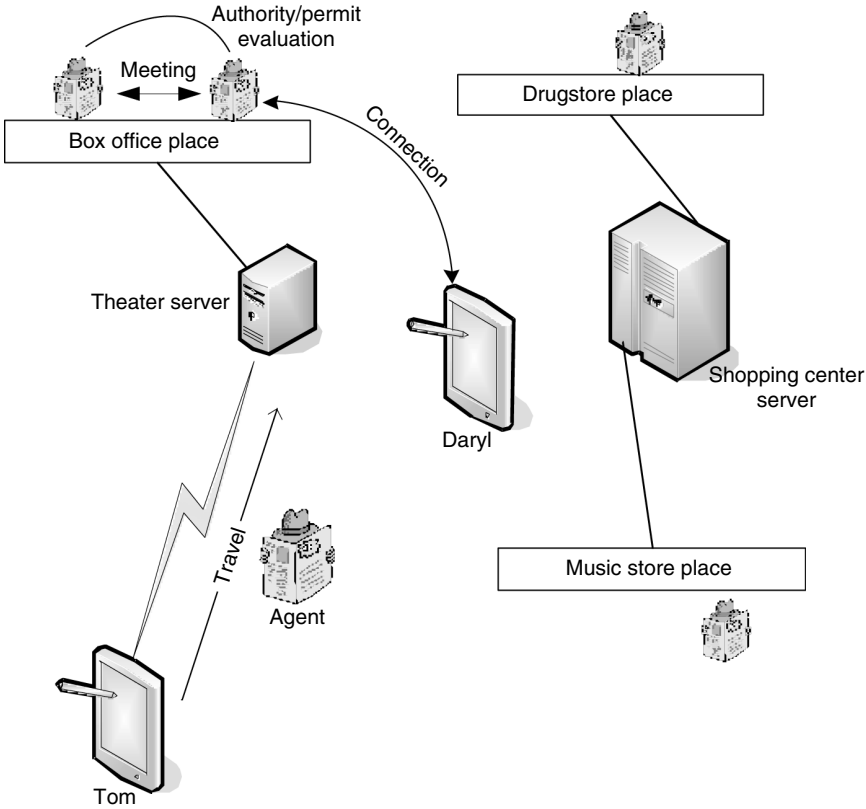

Figure 6.5 The major Telescript components are illustrated above. Tom has just dispatched an agent which has not yet arrived at the theater server. When Tom's agent arrives, it will interact with the static agent in the box office place to arrange for theater tickets. Daryl previously dispatched an agent to purchase tickets and has a connection with her agent in the box office place, so she can actively negotiate prices. Daryl's agent and the box office agent have identified each other through their respective authorities and permits associated with Daryl's agent have been evaluated to see what actions are permitted. The static agents in the drugstore and music store places, which both reside on a shopping center server, are currently idle. To interact with the drugstore or music store agents, Daryl or Tom's agents will have to travel to the drugstore and music store places.

offer a "place" generally will not be visitable by agents. Places offer agents a resting spot in which they can access resources local to that place through a stationary agent that "lives" there, interacting with other agents currently visiting that place.

Travel. Travel allows agents to move closer to or to colocate with needed resources. For example, an agent dispatched by a user to obtain tickets to a jazz concert and reservations at one of several restaurants (depending on availability) might travel from its home place to the place hosted

by the jazz club's box office before traveling to the places hosted by the restaurants. The primary difference between mobile code strategies such as Java applets and agents is that agents travel with at least part of their state intact—after travel, an agent can continue the computation it was engaged in at the instant that travel was initiated. Migration is studied in further detail below in the section entitled, "Migration Strategies."

Meetings. Meetings are local interactions between two or more agents in the same *place*. In Telescript, this means that the agents can invoke each other's procedures. The agent in search of jazz tickets and a restaurant reservation (discussed under "Travel" above) would engage in meetings with appropriate agents at the ticket office and at the restaurant's reservation office to perform its duty.

Connections. Connections allow agents at different places to communicate and allow agents to communicate with human users or other applications over a network. An agent in search of jazz tickets, for example, might contact the human who dispatched it to indicate that an additional show has been added, although the desired show was sold out (e.g., "Is the 11 P.M. show OK?"). Connections in Telescript require an agent to identify the name and location of the remote agent, along with some other information, such as required quality of service. This remote communication method, which tightly binds two communicating agents (since both name and location are required for communication), is the most restrictive of the mechanisms discussed in further detail below in the section entitled, "Communication Strategies."

Authorities. An agent's or place's authority is the person or organization (in the real world) that it represents. In Telescript, agents may not withhold their authority; that is, anonymous operation is not allowed—the primary justification for this limitation is to deter malicious agent activity. When an agent wishes to migrate to another location, the destination can check the authority to determine if migration will be permitted. Similarly, an agent may examine the authority of a potential destination to determine if it wishes to migrate there. Implementation of authorities in an untrusted network is nontrivial and requires strong cryptographic methods because an agent's authority must be unforgeable.

Permits. Permits determine what agents and places can do—they are sets of capabilities. In general, these capabilities may have virtually any form, but in Telescript they come in two flavors. The first type of capability determines whether an agent or place may execute certain types of instructions, such as instructions that create new agents. The second type

of capability places resource limits on agents, such as a maximum number of bytes of network traffic that may be generated or a maximum lifetime in seconds. If an agent attempts to exceed the limitations imposed by its permits, it is destroyed. The actions permitted an agent are those which are allowed by both its internal permits and the place(s) it visits.

Other issues. A number of details must be taken into account when designing an architecture to support mobile agents, but one of the fundamental issues is the choice of language for implementation of the agents (which might differ from the language used to implement the agent *architecture*). To support migration of agents, all computers to which an agent may migrate must share a common execution language. While it is possible to restrict agents to a particular computer architecture and OS (e.g., Intel 80×86 running Linux 2.4), clearly, agent systems that can operate in heterogeneous computer environments are the most powerful. Compiled languages such as C and C++ are problematic because agent executables must be available for every binary architecture on which agents will execute. Currently, interpreted languages such as Java, TCL, and Scheme are the most popular choices because many problems with code mobility are alleviated by interpreted languages. In cases where traditionally compiled languages such as C++ are used for implementation of agents, a portable, interpreted byte code typically is emitted by a custom compiler to enable portability (e.g., see Gray et al., 2000). Java is particularly popular for mobile agents because Java has native support for multithreading, object serialization (which allows the state of arbitrary objects to be captured and transmitted), and remote procedure calls.

Other factors, aside from the implementation language for agents, include migration strategies, communication, and security. Migration and communication strategies are discussed in detail below. A thorough treatment of security in agent systems is beyond the scope of this chapter.

6.2.3 Migration strategies

To support the migration of agents, it must be possible to either capture the state of an agent or to spawn an additional process that captures the state of the agent. This process state must then be transmitted to the remote machine to which the agent (or its child, in the case of spawning an additional process) will migrate. This is equivalent to process checkpointing, where the state of a process, including the stack, heap, program code, static variables, etc., is captured and stored for a later resuscitation of the process. Process checkpointing is a very difficult problem that has been studied in the operating systems and distributed systems communities for a number of years, primarily to support fault

tolerance and load balancing (Jul et al., 1988; Douglis and Ousterhout, 1991; Plank, 1995). In general, commodity operating systems do not provide adequate support for checkpointing of processes, and add-on solutions [e.g., in the form of libraries such as libckpt (Plant, 1995)] are nonportable and impose significant restrictions, such as an inability to reconstitute network connections transparently. A number of research operating systems have been designed that better support process migration, but since none of these is viable commercially (even in the slightest sense), they are not currently appropriate platforms.

Checkpointing processes executing inside a virtual machine, such as Java processes, are a bit easier, but currently most of these solutions (Richard and Tu, 1998; Sakamoto, Sekiguchi, and Yonezawa, 2000; Truyen et al., 2000) also impose limitations, such as restrictions on the use of callbacks, network connections, and file activity. The virtual machine itself can be checkpointed, but then the issues of portability discussed earlier reemerge, and network connections and file access will still pose problems. So where is this going? The punch line is that if commodity operating systems are to be targeted by agent systems—and for wide-scale deployment, this must be the case—then completely capturing the state of general processes to support migration is rife with problems.

One solution is to impose strong restrictions on the programming model used for mobile agents. Essentially, this entails capturing only the essential internal state of an agent, i.e., sufficient information about its current execution state to continue the computation on reconstitution, combined with a local cleanup policy. This means that an agent might perform a local cleanup, including tearing down communication connections and closing local files, before requesting that the agent middleware perform a migration operation. For example, in Aglets (Lange and Oshima, 1998), which is a Java-based mobile agents system, agents are notified at the beginning of a migration operation. It is the responsibility of an individual agent, on receiving such a notification, to save any significant state in local variables so that the agent can be properly "reconstituted" at the new location. Such a state may include the names of communication peers, loop indices, etc. Agent migration in Aglets begins with an agent initiating a migration (its own or that of another agent) by invoking `dispatch()`. A callback, `onDispatch()`, will be triggered subsequently, notifying the agent that it must save its state. After the migration, the agent's `onArrival()` callback will be invoked so that the agent can complete its state restoration.

6.2.4 Communication strategies

Communication among agents in a mobile agent system can take many forms, including the use of traditional CS techniques, remote procedure

call, remote method invocation (e.g., using Java's RMI), mailboxes, meeting places, and coordination languages. Each of these communication strategies has advantages and disadvantages, some of which are exacerbated in mobile agent systems. One consideration is the degree of temporal and spatial locality exhibited by a communication scheme (Cabri, Leonardi, and Zambonelli, 2000).

Temporal locality means that communication among two or more agents must take place at the same physical time, like a traditional telephone conversation. Interagent communication mechanisms exhibiting temporal locality are limiting in a mobile agent's architecture because all agents participating in a communication must have network connectivity at the time the communication occurs. If an agent is in transit, then attempts to communicate with that agent typically will fail.

Spatial locality means that the participants must be able to name each other for communication to be possible—in other words, unique names must be associated with agents, and their names must be sufficient for determining their current location. Some of the possible communication mechanisms for interagent communication are discussed below.

Traditional CS communication. Advantages of traditional CS mechanisms such as sockets-based communication, Remote Method Invocation (RMI) in Java, and CORBA include a familiar programming model for software developers and compatibility with existing applications. Significant drawbacks include strong temporal and spatial locality—for communication to be possible, agents must be able to name their communication peers and initiate communication when their peers are also connected. RMI and other communication mechanisms built on the Transmission Control Protocol/Internet Protocol (TCP/IP) also require stable network connectivity; otherwise, timeouts and subsequent connection reestablishments will diminish performance significantly. Examples of agents systems that use traditional CS mechanisms are D'Agents (Gray et al., 1998) and Aglets (Lange and Oshima, 1998). In Aglets, an agent first must obtain another agent's proxy object (of type AgletProxy) before communication can take place. This proxy allows the holder to transmit arbitrary messages to the target and to request that the target agent perform operations such as migration and cloning (which creates an identical agent). To obtain a proxy object for a target agent, an agent typically must provide both the name of the target agent and its current location. If either agent moves, then the proxy must be reacquired.

Meeting places. Meeting places are specific places where agents can congregate in order to exchange messages and typically are defined statically, avoiding problems with spatial locality but not temporal locality. In Ara (Peine and Stolpmann, 1997), meeting places are called *service*

points and provide a mechanism for agents to perform local communication. Messages are directed to a service point rather than to a specific agent, eliminating the need to know the names of colocated agents.

Tuple spaces. Linda-like tuple spaces are also appropriate for interagent communication. Linda provides global repositories for tuples (essentially lists of values), and processes communicate and coordinate by inserting tuples into the tuple space, reading tuples that have been placed into the tuple space, and removing tuples from the tuple space. Tuple spaces eliminate temporal and spatial bindings between communicating processes because communication is anonymous and asynchronous.

Retrieval of objects from a tuple space is based on the content or type of data, so choosing objects of interest is easier than if objects were required to be explicitly named. One agent architecture that provides a Linda-like communication paradigm is Mobile Agent Reactive Spaces (MARS; Cabri, Leonardi, and Zambonelli, 2000). MARS extends Java's JavaSpaces concept, which provides `read()`, `write()`, `take()`, `readAll()`, and `takeAll()` methods to access objects in a JavaSpace. The extensions include introduction of reactions, which allow programmable operations to be executed automatically on access to certain objects in an object space. This allows, for example, a local service to be started and new objects introduced into the object space, all based on a single access to a "trigger" object by an agent. While distributed implementations of the Linda model exist, MARS simply implements a set of independent object spaces, one per node. Agents executing on a particular node may communicate through the object space, but agents executing on different nodes cannot use the object spaces to communicate directly. MARS is intended as a communication substrate for other mobile agent systems rather than as an independent mobile agent system.

6.3 Summary

This chapter introduced two types of middleware, adaptation middleware and mobile agent systems. Adaptation middleware assists applications in providing the best quality of service possible to users, given the widely fluctuating resource levels that may exist in mobile environments. Mobile agents provide an alternative to static client/server systems for designing interesting mobile applications that access remote data and computational services. Rather than issuing remote procedure calls against distant services, mobile agents migrate code closer to the action to reduce communication and computational requirements for mobile hosts. When a mobile agent has completed its tasks, it can then return home to present the results to the user (or to

another application). Both of these types of middleware are complementary to service discovery frameworks, which are the subject of the next chapter.

6.4 References

"Advanced Configuration and Power Interface," *http://www.acpi.info/*.

Ahuja, S., N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer* 19(8):26, 1986.

Bharat, K. A., and L. Cardelli, "Migratory Applications," in *Proceedings of the Eighth Annual ACM Symposium on User Interface Software and Technology*, November 1995.

Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* 2d ed. O'Reilly, 2002.

Bradshaw, J. M., M. Greaves, H. Holmback, W. Jansen, T. Karygiannis, B. Silverman, N. Suri, and A. Wong, "Agents for the Masses?" IEEE Intelligent Systems, March–April 1999.

Cabri, G., L. Leonardi, and F. Zambonelli, "MARS: A programmable coordination architecture for mobile agents," *IEEE International Computing* 4(4), 2000.

de Lara, E., D. S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-Based Adaptation for Mobile Computing," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (USITS-01), Berkeley, CA, March 2001.

Douglis, F., and Ousterhout J., "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software: Practice and Experience* 21(8), August 1991.

Efstratiou, C., A. Friday, N. Davies, and K. Cheverst, "A Platform Supporting Coordinated Adaptation in Mobile Systems," in *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02).* Callicoon, NY: U.S., IEEE Computer Society, June 2002, pp. 128–137.

Efstratiou, C., A. Friday, N. Davies, and K. Cheverst, "Utilising the Event Calculus for Policy Driven Adaptation in Mobile Systems," in Lobo, J., Michael, B. J., and Duray N. (eds): *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002).* Monterey, CA., IEEE Computer Society, June 2002, pp. 13–24.

Fuggetta, A., G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering* 24(5):May 1998.

Gray, R. S., "Agent Tcl: An Extensible and Secure Mobile-Agent System," in *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96),* Monterey, CA, July 1996.

Gray, R. S., "Agent Tcl: An Extensible and Secure Mobile-Agent System," PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997.

Gray, R. S., D. Kotz, G. Cybenko, and D. Rus, "D'Agents: Security in a Multiple-Language, Mobile-Agent System," *Mobile Agents and Security*,1419:154–187, 1998.

Gray, R. S., D. Kotz, G. Cybenko, and D. Rus, "Mobile Agents: Motivations and State-of-the-Art Systems," Dartmouth Computer Science Department Technical Report TR2000-365, 2000.

Hjalmtysson, H., and R. S. Gray, "Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program," Proceedings of the 1998 USENIX Technical Conference, 1998.

Joseph, A. D., J. A. Tauber, and M. Frans Kaashoek, "Mobile Computing with the Rover Toolkit," *IEEE Transactions on Computers: Special Issue on Mobile Computing*, 46, March 1997.

Jul, E., H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems* 6 1, February 1988.

Keeney, J., and V. Cahill, "Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework," in *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, 2003.

Kowalsky, R. "A Logic-Based Calculus of Events," *New Generation Computing* 4:67, 1986.

Lai, K., and M. Baker, "Measuring Bandwidth," in *Proceedings of IEEE Infocom'99*, March 1999.

Lange, D., and M. Oshima: *Programming and Deploying Java Mobile Agents with Aglets*, Reading, MA, Addison Wesley, 1998.

Litzkow, M., and M. Solomon, "Supporting Checkpointing and Process Migration Outside the Unix Kernel," in *Proceedings of the 1992 Winter USENIX Technical Conference*, 1992.

Milojicic, D., M. Breugst, I. Busse, et al, "MASIF: The OMG Mobile Agent System Interoperability Facility," in *Proceedings of the International Workshop on Mobile Agents (MA'98)*, Stuttgart, September 1998.

Milojicic, D., F. Douglis, and R. Wheel (eds), *Mobility: Processes, Computers and Agents.* ACM Press, 1999.

Nahrstedt, K., D. Xu, D. Wichadukul, and B. Li, "QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments," *IEEE Communications* 39(11): 140–148, November 2001.

Noble, B., "System Support for Mobile: Adaptive Applications," *IEEE Personal Computing Systems* 7(1):44, 2000.

Noble, B., M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile Application-Aware Adaptation for Mobility," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, 5–8 Oct. 1997, pp. 276–287.

Peine, H. "Security Concepts and Implementations for the Ara Mobile Agent System," in *Proceedings of the Seventh IEEE Workshop on Enabling Technologies: Infrastructure for the Collaborative Enterprises*, Stanford University, June 1998.

Peine H., and T. Stolpmann: "The Architecture of the Ara Platform for Mobile Agents," in *Proceedings of the First International Workshop on Mobile Agents (MA '97),* Vol. 1219 of *Lecture Notes in Computer Science.* Berlin, Springer, 1997.

Plank, J. "Libckpt: Transparent Checkpointing under Unix," in *Proceedings of the Usenix Winter 1995 Technical Conference*, New Orleans, January 1995.

Ranganathan, M., A. Acharya, S. Sharma, and J. Saltz, "Network-Aware Mobile Programs," in *Proceedings of the 1997 USENIX Technical Conference*, 1997, pp. 91–104.

Richard, G. G., III, and S. Tu, "On Patterns for Practical Fault Tolerant Software in Java," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 144–150.

Sakamoto, T., T. Sekiguchi, and A. Yonezawa, "Bytecode Transformation for Portable Thread Migration in Java," in *Proceedings of the Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA),* September 2000, pp. 16–28.

Truyen, E., B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java," in *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA).* September 2000, pp. 29–43.

Walsh, T., N. Paciorek, and D. Wong, "Security and Reliability in Concordia," in *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences,* vol. 7. January 1998.

Wong, D., N. Paciorek, and D. Moore, "Java-Based Mobile Agents," *Communications of the ACM* 42(3):XX, March 1999.

Wong, D., N. Pariorek, T. Walsh, et al. "Concordia: An Infrastructure for Collaborating Mobile Agents," in *Proceedings of the First International Workshop on Mobile Agents (MA '97),* Vol. 1219 of *Lecture Notes in Computer Science.* Berlin, Springer, 1997.