TOWARD UBIQUITOUS ENVIRONMENTS FOR MOBILE USERS

A computational paradigm that lets processes migrate with users would support truly ubiquitous computing environments. This article investigates the technical problems that must be solved to accommodate this model and examines some systems that are addressing the problems now.

ALAN DEARLE
University of Stirling, Scotland

In the past 50 years, we have become used to a computation model in which programs are executed on a single fixed piece of hardware and users interact with them through a single fixed display device. With this model, information tends to become inaccessible when the user changes location—even if the user is fully connected to the original host. For example, consider data held in editor buffers and word processors, variables set in shells, information locked in personal information managers, and mail messages retained in electronic mail systems. In each case, the dynamic data associated with the user's interaction becomes inaccessible. In some cases, this is because the data is ephemeral—held in the virtual memory of the processes with which the user interacts. In other cases, the data is in the file store but locked by the processes accessing it.

Java has changed this paradigm a little. It is now common to load a program from a remote site and run it locally, perhaps interacting with the site from which it was originally loaded. However, the display device and computer on which the program executes remain fixed.

Increasing hardware power and high-speed network availability, together with the ubiquitous nature of the Internet, are fueling a radical paradigm shift to a new computational model. In this model, processes are free to migrate with users and thereby provide ubiquitous computing environments. This article investigates the technical

1089-7801/98/$10.00 ©1998 IEEE

problems that must be solved to accommodate this model and surveys some systems that are addressing them.

SOME TERMINOLOGY

Figure 1 illustrates three classes of mobile entities: users, views and platforms. For our purposes here, a user is a person who uses a computer. Users are mobile; they move from home to the workplace, from city to city and continent to continent. A view is what users see when they sit down at a display screen, whether they are interacting with a personal digital assistant (PDA) or a mainframe. Users may own multiple views but only use one at a time. A view includes the user's environment and interfaces to applications that may be executing locally or elsewhere on the network.

A view is implemented by a platform, defined as a collection of hardware and software. The platform software consists of

- active threads (including processes) that implement the view,
- the code being executed by the threads,
- the code that implements the software environment (for example, the Java virtual machine), and
- the data representing entities visible in the view (for example, the process heap and stacks).

The platform hardware consists of a computation environment: a CPU and main memory, a screen with a pointing device, and perhaps a keyboard; it may contain nonvolatile storage but need not. In cases where persistent storage is not available on the platform, it is provided by a server that may be used as a general-purpose repository. There are no requirements for a server to support any form of view.

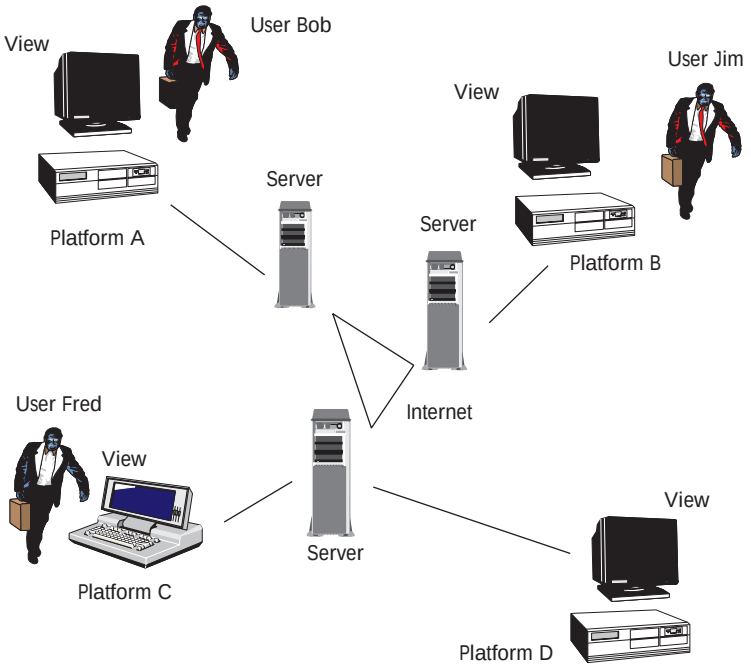

Figure 1. A framework for mobility: Users, views, platforms, and servers.

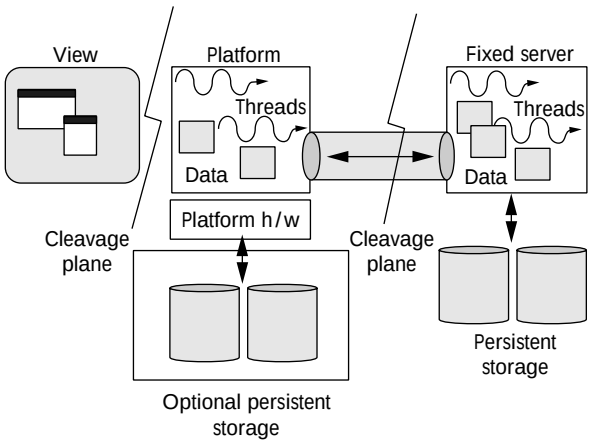

Figure 2. Points of mobility are in the cleavage planes: view mobility over platform and platform mobility over fixed server.

Characterizing Mobility

When users move from location to location, they require their view to migrate with them. For example, in Figure 1, user Jim may move from platform B to platform D, which connects to a different server. When Jim moves, his view should also move, permitting him to continue whatever work he was performing at platform B.

A platform may also migrate with a user, for example, when a user carries a laptop to another site. When this occurs, the applications running on the platform and the view it presents also migrate. However, network connections to the platform are likely to be severed and must be reconnected at the new site—a situation analogous to migrating a process to another platform. When a platform migrates, it could employ the services of a local server at the new site.

Figure 1 is refined in Figure 2, which shows the composition of views, platforms, and servers. The term cleavage plane refers to those places where one entity is mobile with respect to another. Figure 2 shows two cleavage planes that must be considered: (1) view mobility over platforms and (2) platform mobility over servers.

View mobility is supported by platforms and servers, whereas platform mobility is supported only by servers. When a view migrates from one platform to another, either the threads and data implementing the view must also migrate, or the threads must be notified of the location of

the new view and the I/O must be redirected to and from the new platform.

Whether it is better to move the executing threads or to notify the threads of a new view location is a complex trade-off among the utilization of network bandwidth, latency of operations, utilization of local and remote CPUs (computational load balancing), and implementation complexity. Both solutions require the use of network bandwidth but at different times. Moving the executing threads requires migrating at least part, and potentially all, of the process closure to the new platform ("closure" is used here to mean that all states are reachable from the process entity).

Some of this migration can be performed on demand when and if it is required. At first this might seem to be a poor solution since the process closure is potentially large. However, provided that the closure can be appropriately marshaled, it may be copied in large blocks to make more effective use of network bandwidth. Once you have established a working set on the new platform, the network usage drops to almost zero.

The approach of redirecting I/O to alternative devices also has a moderate initial bandwidth requirement since data must be sent to the new platform to initialize the display. However, since the display and the applications interacting with it are now physically separated, all user I/O must be sent to the remote platform and potentially echoed back. This makes poor use of modern networks since the amount of data in each packet is typically very small (often a single character or mouse event). The resulting network traffic is therefore likely to be higher than in the first case.

The second solution also suffers from problems with latency. Although modern networks provide high bandwidth, the laws of nature impose a fundamental delay between send and receive events. Therefore, if any distance separates the old and new platforms, latency in keyboard and mouse feedback becomes intolerable. This is proven by the simple experiment of physically separating an X Windows display from a Unix host by any significant distance.

The first approach fully utilizes the increasingly significant CPU power of the platform. In the second approach, the local CPU merely (1) manages the screen and (2) vectors the keyboard and mouse interrupts to the remote platform. We can therefore conclude that moving threads and data to facilitate view migration is the superior approach. However, this solution is considerably more complex than merely redirecting I/O to a different device. It is likely that hybrid solutions will be developed supporting both solutions and permitting the most appropriate alternative to be employed. However, in the remainder of this article, we'll assume that threads and data implementing a view migrate with the user, and examine strategies that may be used to overcome the associated technical problems.

CLOSURE MIGRATION

Some technical problems are intrinsic to all mobile solutions, for example, the identification and authentication of users. However, a number of problems are particular to the migration of process closures. These include environment mobility, channel mobility, migrating code, migrating state, and locating views. We will review each of these in turn in this section.

Environment Mobility

Environment mobility is concerned with bindings between threads and the external environment. For example, threads may make use of a printer or some input device. When the thread migrates to a remote platform, action must be taken to accommodate this binding. The possibilities include setting the binding to null, re-establishing a binding to an equivalent local resource, and maintaining a remote binding to the remote resource.

Flexible binding mechanisms must be provided so that migratory applications running on a mobile platform may bind to services that appear in their local environment. Furthermore, even for one class of service, different binding regimes are required. For example, a mobile user may wish to print confidential documents on a secure printer in a fixed location. Here, the binding between the application and the external service is static. In other situations, any local postscript printer may be suitable and the binding is dynamic.

In both cases, the external services are provided by servers; the only issue is how platforms bind to the servers. Clearly, if either the hardware or software platform is permitted to migrate, some indication of the (re)binding regime must be specified when the binding is initiated. To support (re)binding activities, servers must provide an associative lookup mechanism similar to that provided by CORBA binding servers.1

Channel Mobility

Channel mobility is a special case of environment mobility. A thread running on a platform may open a communications channel with another thread running on a different platform or server. If the platform is taken off-line and moved to another location, the channel may be lost. Like environment mobility, this situation also arises when views are migrated since the threads implementing the view are migrated.

To make movement transparent requires software that maintains the channel during relocation. This may be achieved by either implementing software at both ends of the channel to manage transparent connection/reconnection or using a server as a connection proxy.

Half sessions. The connection and reconnection of channels can be accommodated through an abstraction called a half session. The name, inspiration, and thinking behind this abstraction originates in the seminal work on optimistic

recovery by Strom and Yemini.2 As shown in Figure 3, each platform or server implementing a relocatable channel uses a half session to manage the connection. Output and input half sessions on different machines combine to implement a reliable stream abstraction that can be disconnected and reconnected to different platforms and servers.

Sender half sessions maintain a log of outgoing messages, which is used to recover lost messages. Each outgoing message is accompanied by a (sequence number, incarnation number) pair. The sequence number is a counter that is incremented on every message send. The incarnation number is incremented immediately after every recovery. In a fault-tolerant system, this happens after a failure; in a mobile context, after migration. Following a migration, an application is recovered to the state of its last checkpoint. This state includes its own last incarnation number (which is incremented) and the expected sequence and incarnation number for each channel.

Following a recovery, the receiver informs the sender of all logged messages to avoid retransmission. When a message is received and the state number is less than or equal to that expected, an error has occurred. If the incarnation number is higher than expected, the sender has restarted. If it is lower, the message is a duplicate sent during recovery and may be discarded. If the sequence number is higher than expected, the receiver has failed and the receiver must recover the lost message(s).

To avoid infinite buffering, the protocol includes a mechanism informing the remote site of messages included in snapshots. The receipt of such a message permits the sender to discard logged messages.

Channel Proxies. The half-session abstraction permits two threads running on different servers or platforms to communicate with each other and permits migration of either end of the channel. However, views may be required to communicate with legacy systems that do not implement the channel abstraction. For example, a user may be interacting with a legacy application running on a Unix shell.

This case requires an additional mechanism that permits the platform (hardware, software, or both) to migrate. This may be achieved using the server as a fixed proxy for communication. In such a scheme, the fixed server communicates with the remote party on behalf of the platform. This communication may be implemented using a traditional socket interface. The platform in turn communicates with the server by using the half-session abstraction. The platform can thus be relocated without knowledge of the remote party.

The problems of channel mobility are especially acute when dealing with legacy systems, which—in the mobile context—means almost all current operating systems and environments. Consider a mobile telnet session that is communicating with a Unix host. When the application moves, the migrating application will hold the address of the remote Unix host, but the host will not recognize the migratory application as that with which it was communicating. This problem is being addressed by the mobile-IP standardization initiative.

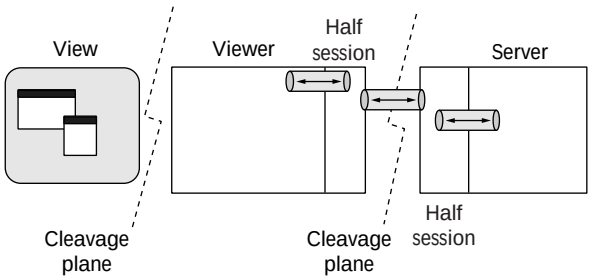

Figure 3. Half sessions.

Migrating Code

Heterogeneity poses problems in all distributed systems, but the problems are especially acute in mobile systems where code and data migrate around a network. The major problem with executable code is that each hardware architecture requires instructions to be encoded in an architecture-specific manner. There are several solutions:

- Translate code from one architecture-specific format to another on migration.
- Provide an architecture-neutral format and a run-time environment on each architecture.,
- Provide an architecture-neutral format and perform just-in-time compilation to the native-code format for each architecture.

The first approach suffers from the N ¥ M problem: Every platform must provide a translator to convert the code from every other architecture into the local format. This is clearly infeasible.

Interpretation by a runtime system. The second approach is that followed by Java: A single architecture-neutral code format is provided and interpreted by a run-time system for each platform. This effectively isolates the code from machine specifics. For example, the Java virtual machine3 is defined as a set of byte-coded instructions with zero or more operands that execute on a simple stack machine. The machine provides per-thread execution stacks and a single heap shared by all threads. Java programs in byte-coded form are loaded and executed by the interpreter.

Besides the obvious advantages of machine independence, this approach includes a code verifier that checks all programs before execution to ensure that the code represents a legal Java program. The checking performed by the verifier includes targets of all control-flow jumps, legal modification of stacked values, and integrity of the instruction stream.

In the context of mobile process closures, this kind of verification is important for two reasons. First, the mere existence of a verifier implies the existence of a semantic model that defines what constitutes a legal program. Second, it ensures that the run-time environment executes semantically meaningful programs. In the case of Java, this semantic model includes the format of data objects, which guarantees that a strong type regime is being followed. Without such guarantees, the interpretation of data closures and, therefore, many of the techniques described below are all but impossible.

This approach is not without its drawbacks. First, implementing a high-performance abstract machine is a major engineering task. Second, the design of the abstract machine must avert many pitfalls. For example, the current Java virtual machine specification inherently addresses a 32-bit machine. This causes many problems in porting the machine to 16- or 64-bit architectures.

JIT compilation. The third approach to code heterogeneity—providing an architecture-neutral format and compiling down to an architecture-specific format—avoids the N ¥ M problem by providing a single compilation target architecture. It can be efficiently executed because it is in native format, and it has the further advantage of being programming-language independent.

Generating native code from an intermediate form inevitably results in different numbers of instructions being generated on disparate architectures. For example, on a RISC architecture such as Alpha/AXP, more instructions would be generated for each high-level instruction than would be necessary for a CISC architecture such as X86. Thus, the native code streams generated for two separate architectures are likely to have little correspondence. This results in the problem that a computation running on one architecture cannot be suspended at an arbitrary point and restarted on another.

There are two basic solutions to this problem: roll the computation forward or roll it back to a unique point. Both solutions are based on the unique points within all generated code streams that correspond to points in the stream from which they were generated. Such points are sometimes called bus-stops, and can be supported using techniques similar to those used by source-level debuggers to map from source-code line to instruction-stream position and vice-versa.

Architecture-neutral code formats vary considerably—from high-level code that approaches source code in complexity, to low-level code that is similar to real machine code. The checking performed by the Java runtime system is possible partly because Java byte-code is little more than a compressed form of Java source code. At the other extreme, low level RISC-like code has been proposed as an intermediate form.4

High-level formats make it easier to check the semantic integrity of the program and are generally smaller than RISC code sequences (which may be important when shipping code). Correspondingly, low-level code formats are generally larger than interpretable code but require less work when the code arrives at a platform since much of the code generation and optimization can be performed early.

The major problem with low-level code is the extreme difficulty of checking the program's integrity or analyzing what it is doing. Object creation is a good example: In a system with high-level intermediate code, such as Java, objects are created by an op-code (in Java, the new_quick operation3). Should a system wish to track object creation, this operation requires augmentation with the appropriate instructions. However, in the case of low level code, a complex series of instructions involving loading sizes onto stacks and calling functions needs to be recognized. This is made more difficult by code optimizers that may obscure the high-level operations being performed.

Low-level intermediate codes permit multiple languages to be compiled down to them, which can also cause problems for the implementation of mobile code and data. For example, consider a system that tracks pointers in objects to support the transmission of closures to other platforms. Languages that permit unsafe unions—like C—make it impossible to determine if fields contain pointers. These problems are similar to those of garbage collection. However, in garbage collected systems, fields of objects that might be pointers are assumed to point to objects. The application of this technology to mobile systems might unnecessarily ship extremely large volumes of data from platform to platform.

High-level intermediate languages that permit JIT code generation, provide type information, and support semantic verification appear to have considerable advantages over lower level formats.

Migrating State

The next question is how to migrate dynamic state between platforms. Migrating the state of a computation in migratory systems is equivalent to saving the dynamic state in persistent systems.5,6 In general, there are four approaches:

- manually writing save-and-restore code in every application/applet,

- using (Java) serialization techniques to save and restore,
- providing persistence at the address space level, or
- providing persistence at the virtual machine level.7

The first approach is the traditional solution: Write flattening code for every object class in the system. While this is possible for simple data structures, it becomes unmanageable in complex applications. Its only merit is that highly optimized code may be written for the data types. A serious deficiency is the lack of mechanisms for saving the state of processes per se. This restriction is only overcome by writing processes as finite state machines so that the process state is entirely held in explicit data structures. This approach severely limits the way in which programs may be written.

The second approach, known as pickling, has become popular since the introduction of Java object serialization.8 It permits an arbitrary graph of objects to be marshaled into a stream, but it also has problems. Some are fundamental and others accidents of implementation. First, not all fields of Java objects are written to the stream. In particular, static fields are not serialized by default due to a perceived security breach. Second, the active context (that is, active threads) is not saved. Knowing that threads are not preserved across serialization will inevitably force programmers to write code as finite state machines.

A third problem is that pickling is an inefficient, all-or-nothing approach to saving state. It includes no concept of saving only data that has changed since a particular time, such as the start of a transaction or the last checkpoint. It is possible to avoid this problem by selectively serializing objects, but such an approach makes it easy to violate referential integrity. Nor can serialization be used effectively to save anything other than an entire object closure. In many systems, the object closure may be very large. Thus, the serialization approach would also require the use of techniques such as Farkas' Octopus mechanism[9] or weak pointers (which have now been incorporated in Java).

The third approach to saving state—providing functionality at the virtual address-space level—is followed in the design and implementation of the Grasshopper persistent operating system.[10] Among the benefits of this approach is the ability to make all data in the address space persistent, including the states of threads and of the stacks supporting them. Curiously, this does not help in transmitting state to another machine since data must be in an architecture-neutral format to support heterogeneity.

Many persistent systems7,11,12 follow the third approach to saving state—providing the necessary functionality at the virtual machine level. This approach has many merits in the mobile computing application domain. It also addresses many of the shortcomings of the serialization approach. In the case of Java, providing persistence at a level lower than the programming language permits the (reflective) type system to be broken; consequently, all object fields can be migrated rather than only the public ones. Secondly, since the runtime system has access to object implementations, it is easy to save those objects that have been modified since a previous checkpoint or transaction start.

All of these techniques can gather some approximation to the computation's state. The data formats differ in each case. The manual approach produces data in an ad-hoc format that hinders its use in a general-purpose system. Clearly some standard representation is required to save and store the data. In this respect, object serialization is the best approach, except that it fails to capture the dynamic state of computations. The most complete solution—persistence at the address space level—suffers from problems with heterogeneity, and so the virtual machine approach remains the most promising.

There are, however, two further problems that must be addressed using this approach—namely, locating data on stacks and locating data on the heap.

Stack-based data. To save state at the abstract machine level and migrate a computation to other platforms requires mechanisms capable of interpreting the data resident on the computation stack. This is similar to the requirements of garbage collected languages, but stricter. Garbage-collected languages must differentiate only between pointer and non-pointer data. Mobile computation, if it is to support heterogeneity, must establish the type of all data on the runtime stack so that it can be reconstructed on a different platform.

Body Deck.

This requirement is also similar to source code debuggers, such as gdb, which must interpret runtime stacks to provide debugging information.

There are several solutions:

- generate self-saving code,
- use a tagged stack architecture, or
- generate a static map of the stack layout.

The first approach has been experimented with in a version of the Napier88 system developed at the University of Ade-

Body Deck.

laide.[6,13] The mechanism explicitly encodes source-level procedures as restartable native code functions, which are parameterized with a restart point and return a scalar status value. The restart point indicates where the code should start executing. The first call to a procedure is performed by a C function call with a restart point of zero. The status value indicates whether the procedure executed to completion or encountered some hindrance such as a snapshot.

The restartable native code functions use a stack of activation records. Whenever a native procedure is called, a stack object is created to represent its activation record. This object provides a repository in which data may be preserved across checkpoints. When they start executing, all native code procedures register the address of the object containing the activation record in a global data structure. This ensures that the activation record can be located following a restart.

Before a checkpoint is performed, the generated code must ensure that the entire dynamic call-chain is stored in objects. Each procedure saves its entire state in the corresponding stack frame. The saved state includes a resume address that may be passed to the function when it is restarted. After saving its state, each procedure returns a status value to its caller, indicating that it too should save its state and return the same status value to its caller. Since each executing native function has saved its state in an object and returned, there is no data on the native call stack. The mechanism is therefore architecture independent.

The second approach—using a tagged stack architecture—is extremely expensive computationally since every stack load operation must be accompanied with a load of the tag.

The third approach is followed by most source-code debuggers and requires a map of stack usage to be associated with the executable code. Note that such information is usually only included with code if the appropriate debugging flags have been set during the compilation. If this scheme is used to support mobile code, the information must be included with all code.

The complexity of this approach stems from the potential for changes in the execution stack with every operation. The map must therefore contain enough information to determine the stack contents at every legal program-counter address. This is generally achieved by saving a compilation symbol table for each procedure/method. Each symbol table entry must contain both a start and end address at which the entry is legal. The contents of a stack at any program-counter value may therefore be determined by a simple search over the appropriate symbol table at runtime.

Heap-based data—saving. If an entire process closure is to be saved on one platform and restored on another, the dynamic data resident in the heap must also be saved. To do this requires identifying and marshaling the transitive closure of objects pointed to from the stacks. This means that pointers, objects, and data within objects must be identifiable.

We have already dealt identifying pointers: It is a special case of identifying the types of all data on the runtime stacks.

With regard to objects and data within objects, the former must be self-describing so that their size and field types may be determined at runtime. Like the requirements on stacks, this is a stricter form of that for garbage collection, where only pointers and nonpointers must be differentiated. To cope with heterogeneity, the system must be able to determine the types of all fields.

As an example of self-describing objects, consider Sun Microsystem's current Java virtual machine implementation. Each pointer to an object is represented by a handle containing a pair of pointers. The first points to a table containing the object's methods and to a pointer to an object representing the object's class. The second is a pointer to the actual object allocated in the Java heap.[3] The full type of any object may be determined by examining the class object. Thus, the current implementations of the Java virtual machine contain enough information to permit the transitive closure of heap objects reachable from the stacks to be determined.

The next consideration with heap objects is deciding how often to save data and what data to save. Taking periodic process snapshots has several advantages. First, like incremental garbage collection (provided that incremental algorithms are used), periodic snapshots can bound the amount of time required when a process snapshot is taken. Second, process resilience is a free by-product of making periodic snapshots: Should a process fail, it can be restarted from an earlier snapshot. Finally, periodic snapshots require no initiation by the user who can, consequently, walk away from a platform at any time and upload the process closures onto another platform.

The most common incremental algorithm used to preserve state in persistent systems associates a mark bit with every object. When an object is modified, the mark bit is set and then cleared when the next snapshot of the object is taken. This allows a linear sweep over the heap to determine

which objects have been modified since the last checkpoint and consequently need to be marshaled.

When data structures are copied, care must be taken to preserve the referential integrity of circular data structures. In some state-saving regimes, circular data structures are inadvertently changed from graphs into trees. This can change the semantics of programs that operate over the object graphs. This usually occurs when pickling schemes, such as Java object serialization, are used which do not preserve object identity between individual snapshots.

Heap-based data—restoring. When a process is restored on a platform, either the entire or a partial process closure can be loaded from the remote platform/server. Loading the entire process closure is easier to engineer but may require the user to wait for a relatively long period. World Wide Web users commonly experience such delays waiting for data to load. It is therefore expedient to use incremental loading strategies that permit the process(es) to start running before the entire closure is loaded. If this strategy is employed and the working set is smaller than the entire closure, many objects may never need to be loaded.

This strategy requires solutions to three additional problems:

- providing a mechanism to differentiate between the objects that have been loaded and those that have not,
- providing a mechanism to ensure that an object is loaded at most once to avoid problems with referential integrity,
- defining a protocol that permits unique identification of both the site(s) on which remote objects are stored and the objects stored there.

Several techniques have been used in object-oriented databases and persistent systems to differentiate between local and remote objects. The simplest is to check for residency prior to every object de-reference. For example, in PS-algol systems, every pointer to a nonresident object is denoted by setting the top bit of the pointer; a simple test prior to de-reference can screen for nonlocal objects and activate the faulting mechanism if required. Such tests can have a high impact on performance. Fortunately, they can be optimized. For example, the PS-algol system ensured that all pointers on the stack were to local objects, thereby avoiding many tests.

If the language system is purely object-oriented, a modification of this scheme developed by Moss can be employed.[14] It relies on knowing that all method calls must go through a method table. By implementing ghost objects whose methods contain faulting code, nonresident objects can be faulted without requiring any additional tests. The ghost objects implement a barrier, effectively separating loaded from non-loaded objects.

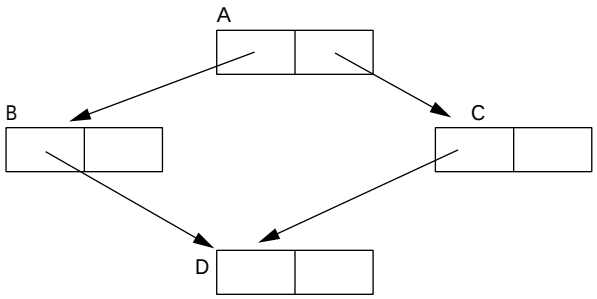

Figure 4. A simple object graph.

Wilson[15] has proposed the implementation of barriers using virtual memory protection mechanisms. His method is similar to that of Moss. The difference is that protected regions of the virtual address space, rather than ghost objects, implement the barrier. When a process attempts to access a nonresident page, the faulting code is activated by the operating system.

Consider the object graph shown in Figure 4. If a breadth-first search from A is performed, the object labeled D will be reached by two independent routes. Consequently, D could easily be loaded twice. To avoid this problem, the runtime system must maintain a map that records loaded objects. Prior to loading any object, this map must be consulted and the local copy of the object used if one is available.

The last problem associated with incremental loading is to define a protocol that permits unique identification of the site(s) and the remote objects. This is easily achieved using a pair containing the IP-address of the site combined with a local identifier for the object. The only complication is that the size of this pair is likely to be bigger than any native pointer size.

Locating Views

The state of processes that implement views may easily be tagged with the user-ID combined with a view identifier. This scheme has been used in the Monads16 and Grasshopper[5] persistent operating systems to identify login sessions.

In Grasshopper, when a user logs out, their environment continues to exist. Rather than create a new environment on each login, users may bind to an extant environment. This task is performed by a login server that maintains a mapping between (username, password) pairs and capabilities for the environments. A similar scheme is described in more detail in Keedy and Vosseberg.[17]

Before a view is made visible, its owner must be authenticated. This requires users to identify themselves and present authentication—just as they do in a conventional login session on a Unix machine. In addition, the location of the view specified by the user must be established. This can be accomplished using smart cards or the Internet.

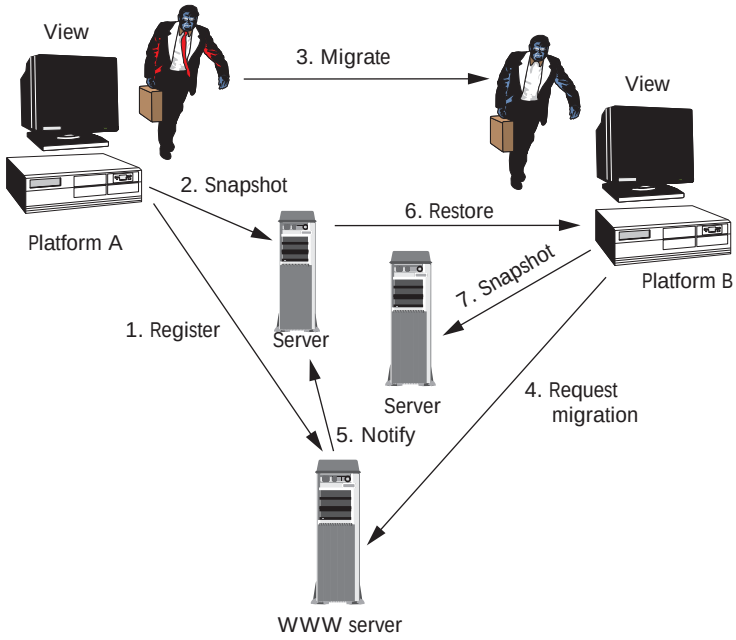

Figure 5. Reestablishing a view.

Smart cards with a small amount of memory can record the identity of the user and where they were last active. Many of the currently available network computers have smart card interfaces built into them that could be used for this purpose. When a user wishes to start using a platform, the smart card would identify the user and determine the platform that they last used. A smart card with more memory could store the roots of the user's active processes, thus allowing computation to start as soon as the user was authenticated. Clearly, if enough memory were available, the user's entire working set could be recorded on the card.

An alternative approach that makes use of the Web assumes each user to have a home capable of recording the identity of the last hardware platform on which each view was last active. This functionality may be implemented via a simple CGI (common gateway interface) script capable of saving and retrieving locations located on a Web server. In the worst case, a user might type in the location of their home during the authentication process so that the system could locate it. Other possibilities include using a search engine or global identifiers to locate a user's home.

As described above, a server may provide the persistent storage of views. When a view is established on a new hardware platform, the user's home is contacted to register the platform's server as the manager of the view as shown in Figure 5, step 1. Periodically, the view is snapshotted to the server to provide resilience and to permit future migration (step 2).

Following user migration (step 3), the new platform contacts the user's home to register and request view migration (step 4). The Web server requests the platform that last activated the user's chosen view to migrate the software platform to the new hardware platform (step 5). These requests are typically sent via the server supporting the platform. In practice, many requests will not be forwarded to the platform and will be handled by the server implementing the platform's persistent storage. In either case, the closure of threads and data implementing the view are migrated to the new hardware platform for restoration (step 6) and the cycle begins again.

EXISTING SYSTEMS

Several systems currently support aspects of mobility described in this article.

Teleporting

The Teleporting System developed by Olivetti Research Laboratory at Cambridge[18] is the best example of supporting mobility without moving the applications. The system lets users interact with their running X Windows applications from any X Windows display device. Teleporting is the act of moving windows from one display and recreating them on another. It is achieved by introducing a level of indirection between applications and their interfaces. A proxy server acts as a proxy for a real X Windows server. Instead of providing a display, keyboard, and mouse, the proxy server diverts I/O to real X-servers. By interacting with the proxy server, the I/O may be diverted to an arbitrary device.

Teleporting is a simple implementation of mobile computing environments and utilizes existing network and workstation infrastructures. The major problem with this approach is that it is unsuitable for anything other than an office environment since it suffers from an intrinsic latency problem.

Migratory Applications

Bharat and Cardelli[19] describe an architecture designed to support migratory application in the language Visual Obliq.[20] Single-user migratory applications are supported at the language-environment level and can migrate from node to node while maintaining the state of their user interface. Almost no requirements are made of the application programmer.

The system's basic building block is an agent, a computation that may hop from site to site carrying with it a suitcase containing the agent's persistent memory. When the agent executes a hop instruction, the suitcase and the computation's closure are migrated to the new site. When an agent arrives at a site, it receives a briefing that may include advice as well as site-specific information.

These mechanisms have been used to construct migratory applications containing a MigrateTo(Host) command

that contacts the remote host and, if the host will accept the application, checkpoints the state of its user interface and performs a hop instruction. This system incorporates many of the concepts described in this article, but there are a few differences. First, when a closure is copied to another site, the mutable values are never copied across the network; instead, network pointers refer to the remote objects. Second, Bharat and Cardelli make no mention of the problems relating to heterogeneity.

Aglets

As described in Lange,[21] an aglet is a mobile Java object capable of visiting different hosts on a network. Aglets are autonomous in that each one contains an active thread of execution and is capable of reacting to messages sent to it. Like an applet, the class files for an aglet can migrate across a network. Unlike applets, when an aglet migrates, it also carries its state. An applet is code that can move across a network from a server to a client whereas an aglet is a running Java program (code and state) that can move from one host to another on a network.

Each aglet executes in a context that provides a uniform execution environment independent of the capabilities of the host and serves to isolate the aglet from the platform. Aglets have an onCreation method that executes when the aglet is created or migrated to a new context. Aglets also contain a dispatch method that takes a URL as a parameter and can be used to migrate the aglet to a new context. When dispatch is called, standard Java object serialization is used to preserve the aglet's bytecode and state and transmit them across the network using the Agent Transport Protocol.[22] Aglets can be reactivated at the new site using the onCreation method.

Since aglets use Java object serialization to export their state, the execution state of the threads owned by the aglet are not serialized. Therefore, an aglet that is migrated or deactivated loses any state resident on stacks and the program counters of running threads. This is because the Java Virtual Machine does not permit direct access to runtime state.

Before an aglet is serialized, the host informs it that serialization is imminent (via the onDispatch method) so that it can preserve any information needed to continue its execution in object variables. This inevitably forces applications to be written in a state-machine-like fashion with consequent loss of expressiveness.

Telescript/Odyssey

Telescript[23] is an object-oriented language developed by General Magic for distributed applications. Its major contribution is a protection mechanism based on capability-like entities called permits. A thread may hold permits, which allow a particular set of operations, for example, to use a resource such as a CPU. The Telescript runtime system is purely interpreted and executes a relatively high-level bytecode. The Telescript architecture has recently been re-implemented in 100 percent pure Java and the system renamed Odyssey.

Facile

Facile was developed at European Collaborative Research Centre.[24] It is Standard ML extended with primitives for concurrency, distribution, and communication. In line with its ML heritage, it models resources as strongly typed sets of functions. Any resource that matches the expected signature may be bound when code migrates to a new platform.

This simple mechanism provides support for environment mobility. Communication between nodes is provided by a channel mechanism capable of transmitting arbitrary Facile values, including function closures. The system supports both interpreted architecture-neutral code and native code using JIT compilation when appropriate.

Omniware

The Omniware system exemplifies a low-level intermediate code approach.[4] It uses low-level RISC-like instruction formats to represent code. The system provides a virtual machine called OmniVM. When virtual machine code is loaded onto a platform, it is compiled down to native code for the host architecture.

Although the system is language independent, it is allegedly safe through a sandboxing technique developed by one of the authors.[25] However, the authors do not mention how integrity is enforced when modules interact. The authors claim that the Omniware approach is general enough to support languages such as C and C++. This is important because, even interpreted systems such as Java and Visual Basic, as much as 90 percent of the code executed is in library routines is written in C and C++.

CONCLUSIONS

As the technologies of high-performance networks and fast CPUs become inexpensive, reliable, and commonplace, the computer is increasingly a common tool for many forms of work and leisure rather than a computing engine. Many workers are now highly mobile and wish to be able to work seamlessly over a number of platforms. These changes motivate a break from the traditional model of computation to an ubiquitous model that makes the user's entire environment available wherever it is required.

This article has examined many implementation issues that must be addressed to engineer such an environment. The review of current systems addressing these issues reveals that it is now possible to engineer a system in which the user's entire environment is available wherever and whenever it is required. However, the provision of such an environment represents a considerable engineering effort and many significant choices. At the University of Stirling, we are cur-

rently investigating these choices in the construction of a ubiquitous environment based on Java. ■

ACKNOWLEDGMENTS

I would like to thank David Hulse and the anonymous referees for helpful comments on drafts of this article.

REFERENCES

1. S. Baker, CORBA Distributed Objects Using Orbix, Addison-Wesley, Harlow, England, 1997.
2. R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," ACM Trans. Computer Systems, Vol. 3, No. 3, 1985, pp. 204-226, 1985.
3. T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, Reading, Mass., USA, 1997.
4. S. Lucco, O. Sharp, and R. Wahbe, "Omniware: A Universal Substrate for Web Programming," Proc. 4th Int'l World Wide Web Conference: The Web Revolution, 1995, http://www.w3.org/Conferences/WWW4/Papers, 1995.
5. A. Dearle et al., "Grasshopper: An Orthogonally Persistent Operating System," Computer Systems, Summer, 1994, pp. 289-312.
6. R. Morrison et al., "The Napier88 Reference Manual," Tech. Report PPRR-77-89, University of St. Andrews, 1989.
7. M.P. Atkinson et al., "Design Issues for Persistent Java: A Type-Safe, Object-Oriented, Orthogonally Persistent System," in Proc. 7th Int'l Conf. on Persistent Object Systems, Workshops in Computing, Springer-Verlag, Berlin, 1996, pp. 33-47.
8. Javasoft, Object Serialization Specification, 1997, http://chatsubo.javasoft.com/current/doc/serial-spec/serialTOC.doc.html.
9. A. Farkas and A. Dearle, "Octopus: A Reflective Language Mechanism for Object Manipulation," Proc. 4th Int'l Workshop on Database Programming Languages, Springer-Verlag, London, 1994, pp. 50-64.
10. J. Rosenberg et al., "Operating System Support for Persistent and Recoverable Computations," Comm. ACM, Vol. 39, No. 9, 1996, pp. 62-69.
11. "PS-algol Abstract Machine Manual," Tech. Report PPRR-11-85, Universities of Glasgow and St Andrews, 1985.
12. A.L. Brown et al., "The Persistent Abstract Machine," Tech. Report PPRR-59-88, Universities of Glasgow and St. Andrews, 1988.
13. S.J. Bushell et al., "Using C as a Compiler Target Language for Native Code Generation in Persistent Systems," Proc. 6th Int'l Workshop on Persistent Object Systems, Workshops in Computing, Springer-Verlag, Berlin, 1994, pp. 164-183. Also available at ftp://persistence.cs.stir.ac.uk/pub/papers/GH-04.ps.Z.
14. J.E.B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle," IEEE Trans. Computers, 1991, pp. 1-39.
15. P. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware," Computer Architecture News, Vol. 19, No. 6, June 1991, pp. 6-13.
16. J. Rosenberg. "The MONADS Architecture—A Layered View," Proc. 4th Int'l Workshop on Persistent Object Systems, Morgan Kaufman, San Francisco, Calif., USA, 1990.
17. J.L. Keedy and K. Vosseberg, "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System," Proc. 25th Hawaii Int'l Conf. on System Sciences, IEEE Computer Society Press, Los Alamitos, Calif., USA, 1992, pp. 747-756.
18. F. Bennett, T. Richardson, and A. Harper. "Teleporting—Making Applications Mobile," in Proc. Workshop on Mobile Computing Systems and Applications, 1994. Available at http://www.cam-orl.co.uk/teleport/teleport.html.
19. K. Bharat and L. Cardelli, "Migratory Applications," Tech. Report 138, DEC SRC, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-138.html, 1996.
20. K. Bharat and M.H. Brown, "Visual Obliq: A System for Building Distributed, Multi-User Applications by Direct Manipulation," Tech. Report 130, Digital Systems Research Center, 1996, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-130a.html.
21. D.B. Lange, "Java Aglet Application Programming Interface," white paper, 2nd draft, 1997, IBM Tokyo Research Laboratory, http://www.trl.ibm.co.jp/aglets.
22. D.B. Lange and Y. Aridor, "Agent Transfer Protocol—ATP/0.1," draft specification, IBM Tokyo Research Laboratory, 1997, http://www.trl.ibm.co.jp/aglets/atp/atp.htm.
23. J.E. White, "MobileAgents," white paper, General Magic Inc., http://www.genmagic.com, 1995.
24. B. Thompson et al., "Facile Antigua Release Programming Language," Tech. report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany, 1993, http://www.ecrc.de/facile/facile_home.html.

Alan Dearle studied for his B.Sc. and Ph.D. at St Andrews University in Scotland where he later became a lecturer. During this time he worked on PS-algol and was a co-designer and implementor of Napier88. In 1990 Dr. Dearle moved to The University of Adelaide, Australia, becoming a reader in 1992. At Adelaide he worked on a distributed implementation of Napier88 and persistent programming environments. In 1992 he was co-founder of the Grasshopper Persistent Operating Systems Project. In 1994 he moved to The University of Stirling to take up a chair in Computing Science, becoming Head of Department in 1988.

URLs for this article

"Mobile IP Resources—Standardization Process," unpublished paper, 1997, www.neda.com/mobileIpSurvey/html/mobileIP_57.html.

General-Magic, "Telescript Language Reference," Tech. Report www.genmagic.com, 1995.

Sun Microsystems, Java Development Kit V 1.1.1, available for download at java.sun.com/products/jdk/1.1.1/index.html, 1997.