Persistent Servers + Ephemeral Clients = User Mobility

Alan Dearle
University of Stirling, Stirling, Scotland
al@cs.stir.ac.uk

Abstract

A large group of computer users are now mobile; they either make use of more than one computer or carry lap-top computers with them. User migration is often hindered by inadequate programming models and architectures. This paper describes an architecture which permits the user's environment to migrate with them. A corner-stone of this architecture is the ability of persistent Java systems to save and restore the state of active computations. This concept is extended to permit computations to be restored on different machines thus permitting a user's environment to migrate. The architecture also addresses the difficult issue of channel mobility between two migratory applications. It is therefore general enough to support arbitrary distributed mobile computations.

1 Introduction

Many computer users make use of more than one computer, for example, it is common to have a computer at work and another at home. Sometimes within the workplace a user may use more than one computer, perhaps in different rooms, buildings or even countries. Users in these situations have been forced to accept that the data they wish to manipulate may be unavailable on the local machine or perhaps available but the appropriate software to manipulate it is not. Software can add to the problem by encapsulating data, making it difficult or impossible to access from other machines. Examples include: data held in editor buffers, CAD designs, electronic appointment programs and electronic mail.

Many people have adopted ad-hoc working practices to accommodate mobility, for example, carrying floppies or a lap-top containing a cached version their current work. Ironically, the use of a lap-top introduces another form of mobility which must be accommodated - that of machine mobility. A common solution to the problem of making data globally available is to store and manipulate data on a central server. The situation is typified by electronic mail. Some users of electronic mail read their mail using tools such as elm, xmh or mailtool which both execute and manipulate mail files on a central server. Alternatively, tools such as Eudora which run on a local workstation may be used. The former solution forces users onto the central resource whereas the latter utilises the local machine but at a cost. Since mailers such as Eudora maintain both read and unread mail folders on the client[†], a user moving to another machine and wishing to access mail has considerable difficulty.

A more desirable situation would be for a user to approach an arbitrary machine and be able to continue performing their work regardless of where they worked last. This paper presents a first step towards the realisation of this ideal. A number of developments have made such an approach possible: the ubiquity of the Internet, the widespread adoption of Java, and the maturity of persistent technologies.

[†] Pop based mailers such as Eudora do allow mail to be left on the server but the users forfeit the ability to organise mail into folders.

The paper is organised as follows: first some terminology and a characterisation of different kinds of mobility is made. This is followed by a statement of the requirements for supporting mobility and a description of an architecture designed to satisfy these requirements. The architecture has two main components, platforms and servers which are described in Sections 3 and 4 respectively. A design to cope with the problems of environment binding and communication channels between mobile entities is addressed in Section 5. Section 6 describes an initial experiment which we have conducted to prototype, test and develop our ideas. Some related work is discussed in Section 7 and Section 8 concludes.

2 Terminology and Requirements for Mobility

2.1 Terminology

In this paper we will consider the mobility of three classes of entity: *people*, *machines* and *processes*. We assume that a mechanism for finding data on the network exists; a naming mechanism such as that provided by Uniform Resource Locators (URLs) will suffice.

We define a *user* to be a person who uses a computer; users are mobile: they move from home to their place of work, from city to city and from continent to continent. We define a *view* to be that which a user sees when they sit down at a computer screen be it connected to a PDA, PC, workstation, network computer, or mainframe. Users may own multiple views but only make use of one at a time. A view is implemented by a *platform*. A platform is a collection of hardware and software that combine to implement the view. We will separate a platform into two components the *platform software* and the *platform hardware*. The platform software consists of:

- active threads and/or processes[‡] that implement the view,
- the code being executed by the threads,
- the code that implements the software environment (e.g. the Java-virtual machine, dynamic libraries etc.) and,
- the data representing entities visible in the view.

The platform hardware consists of a computation environment on which to run the platform software i.e. a CPU and main memory, a screen with a pointing device and perhaps a keyboard. The platform hardware may contain persistent storage but need not. In cases where persistent storage is not available on the platform, it is provided by a *server*. A server contains non-volatile storage and may be used as a general purpose repository. There are no requirements for a server to support any form of view. If a device which is used as a server also contains the ability to operate as a platform or vice-versa, it will be considered to be two different entities. Clearly, there is an opportunity for optimisation in this case.

2.2 Characterising Mobility

The framework described above is shown in Figure 1 which shows the three classes of entity that may be mobile: users, views and platforms. When users move from location to location they require their view to move with them. The view includes the user interfaces to applications which may be executing on the platform, on the associated server, or elsewhere on the network. When a view migrates from one platform to another, either the threads and

[‡] We will use the term thread to mean thread/process.

data implementing the view must also migrate, or those threads must be notified of the location of the new view.

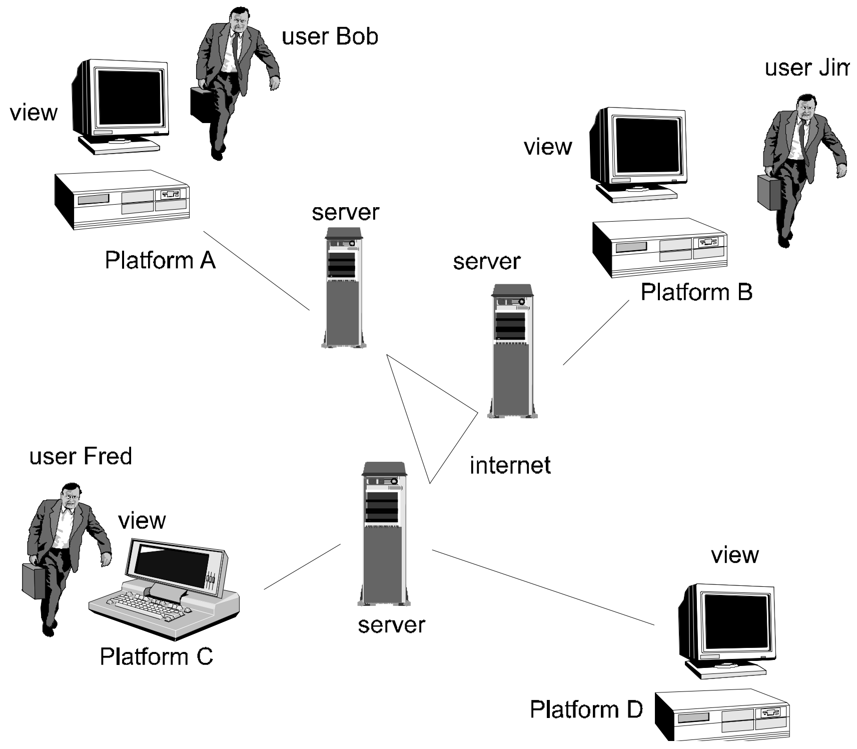

Figure 1: Users, Views, Platforms and Servers.

Clearly the former solution makes better use of caching and network bandwidth. In this paper it is therefore assumed that the threads and data implementing the view migrate with the user. There is no requirement for the applications with which the user is interacting to migrate although it may be expedient for them to do so. When the threads implementing a view migrate, the connections to networked entities must also migrate. For example, if the view includes a traditional window implementing a Unix interaction with a remote host, the input and output to and from that host must migrate with the view.

A platform may migrate with a user, for example, when a user carries a lap-top to another site. When this occurs, the applications running on that platform and the view that it presents also migrate. However, network connections to the platform may be severed and need to be re-connected at another site. This situation is analogous to the process which occurs when platform software migrates. When a platform migrates, it may be expedient for the platform to employ the services of a local server at the new site.

As described above, a view is implemented by a collection of persistent threads each of which operate on some cached data. Figure 1 is refined in Figure 2 which shows the composition of views, platforms and servers.

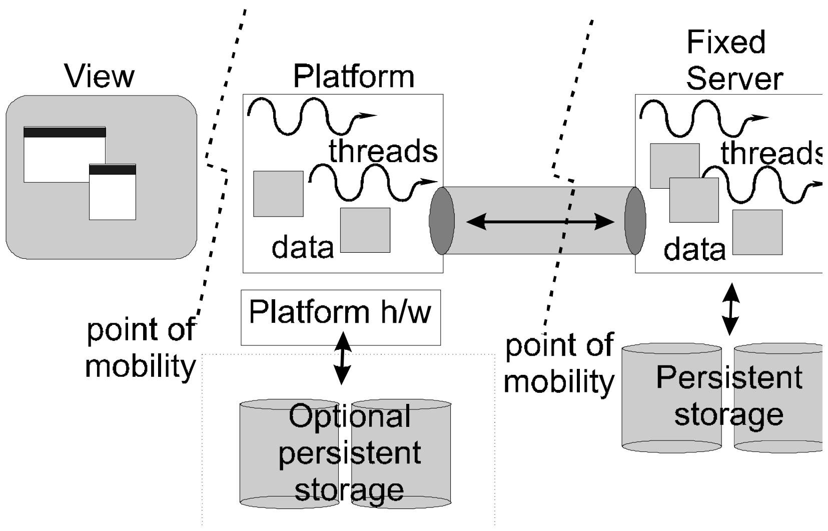

Figure 2: Points of Mobility

Figure 2 shows the two different points of mobility that may be considered:

1. view mobility over platforms and,
2. platform mobility over servers.

View mobility is supported by platforms and servers, platform mobility is supported by servers. The persistence of threads and data in the platform may be implemented either by the platform, if it is equipped with persistent storage (e.g. in the case of a PC), or co-operatively between the platform and the server if it is not (as is the case in a NC).

2.3 Requirements for User Mobility

A user may move from machine to machine, for example, in Figure 1 user *Jim* may move from platform B to platform C connected to a different server. When the user moves, the user's view should also move, permitting the user to continue with whatever work was being performed at the last platform.

The ability to migrate a view requires that the platform software be capable of migrating to a different platform hardware instance. Since the platform software consists of active threads and the data representing entities visible in the view, migrating the view requires two forms of migration:

1. data migration, and,
2. thread migration,

or rephrased,

view mobility = thread mobility + data mobility.

Each of these forms of mobility introduces other problems discussed below.

2.4 Requirements for Platform mobility

Consider platform C, in Figure 1, as a mobile device it may be disconnected from the network and reconnected elsewhere. Here the view of user Fred is (may be) maintained by the mobile device, however the mobility of platforms highlights some other problems, namely:

1. *environment mobility*, and the special case of this,
2. *channel mobility*.

Environment mobility is concerned with bindings between threads and the external environment. For example, a thread may make use of a printer or some input device. Mechanisms must be provided so that threads running on a mobile platform may bind to services which appear in their environment. This situation also arises when a view is migrated to another platform. It is clear that different kinds of bindings are required even for one class of device. For example, in the case of a printer, a user may wish to print confidential or personal documents on a secure printer at a fixed location and in other cases any printer may satisfy their needs. This illustrates the need for both static and dynamic binding mechanisms to support environment mobility.

Channel mobility is a special case of environment mobility. A thread running on a platform may open a communications channel with a another thread running on another platform or server. If the platform is taken off line and moved to another location, the channel will be lost. Like environment mobility, this situation also arises when views are migrated since threads are migrated to other platforms. To make movement transparent, software that maintains the channel across movement must be provided. This may be achieved in one of two ways:

1. implementing software at both ends of the channel to manage the transparent connection/reconnection, and,
2. using a server as a *connection proxy*.

These ideas are expanded in Sections 5.1 and 5.2 below.

2.5 **Overall architecture**

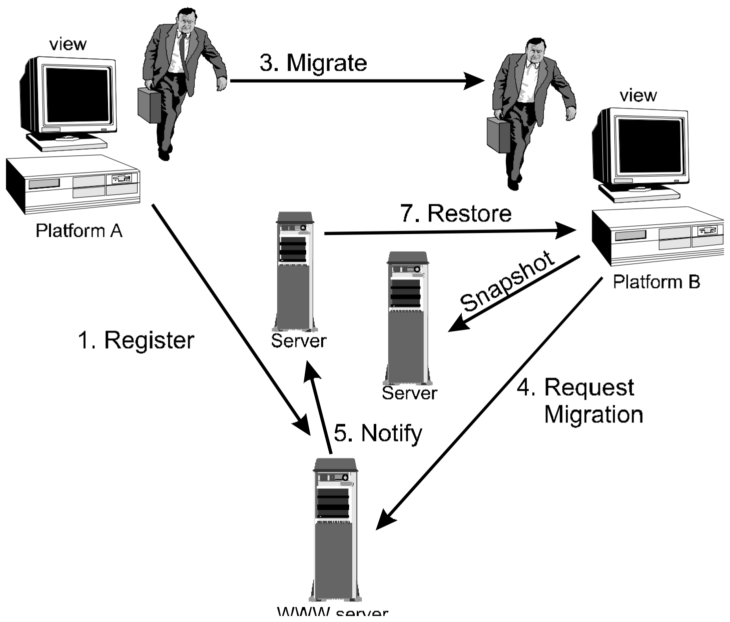

Figure 3: Re-establishing a View

Before view is made visible, the owner of the view must be authenticated. This requires the following:

1. users must identify themselves,
2. users must present authentication, and
3. the location of the view specified by the user must be established.

The first two stages above are identical to a conventional login session on a Unix machine. The third step is necessary in order to locate the user’s specified view. These three steps may be easily achieved if a smart card with modest memory is available. Using smart cards it is possible to record the identity of the user and where they were last active. Many of the Network Computers that are currently available have smart card interfaces built into them which could be used for this purpose. However, since smart card devices are not ubiquitous, we shall consider other ways of identifying the user and their views. Figure 3 shows one method of doing this using the World Wide Web.

Each user is assumed to have a home which is capable of recording the identity of the last hardware platform on which each view was last made. This functionality may be implemented via a simple *cgi* script located on a Web server which is capable of saving and retrieving locations. In the worst case, a user might type in the location of their home during the authentication process in order permit the system to locate it. Other possibilities are to use a search engine or global identifiers to locate a user’s home.

When a view is established on a hardware platform, the user's home is contacted to register the server as manager of the view (1). Periodically the view is snapshotted to the server to provide resilience and to permit future migration (2). Following user migration (3), the new platform contacts the user’s home to register and request view migration (4). The Web server requests the platform which last implemented the user’s chosen view to migrate the software platform to the new hardware platform (5). These requests will typically be sent via the server supporting the platform as described in Section 5.2 below. In practice many of these 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 (6) and the cycle begins again.

3 **Platforms**

A platform must be capable of:

1. authenticating a user,
2. loading a view from the platform/server identified during the authentication sequence, and
3. saving the state of the view to persistent storage provided by the platform or the server that supports it.

The second and third activities are intimately related to each other and require a protocol that defines:

1. how to identify the persistent state implementing a view,
2. how persistent state is preserved,
3. what format the persistent data is in, and
4. how to transport that state to and from persistent storage and between platforms and servers.

The persistent state that implements a view may easily be tagged using the user-id combined with a view identifier. This scheme is used in Grasshopper to identify login sessions and similar schemes have been used elsewhere [1]. The next question is how state is preserved; in general, there are 4 approaches to saving state in Java systems:

1. manually writing save and restore code in every application/applet,
2. perform saving and restoration using (Java) serialisation,
3. providing persistence at the (Java) virtual machine level [2], and

4. providing persistence at the address space level.

The first approach is the traditional solution to persistence: write flattening code by hand for every object class in the system. Whilst this is possible for simple data structures it becomes unmanageable in complex applications and has been estimated to account for 30% of all application code. The only merit of this approach is that code may be written that is highly optimised for the data types.

The second approach has become popular since the introduction of Java object serialisation [3] which permits an arbitrary graph of objects to be marshalled into a stream. Whilst this approach would be appear to be a panacea it is not without its problems some of which are fundamental and others accidents of implementation. The first problem, which may be argued to fall into either of the above two categories, is that not all fields of objects are written to the stream. In particular, private fields are not serialised due to a perceived security breach. The second problem with this approach is that active context (i.e. active threads) is not saved. Knowing that threads are not preserved across serialisation will inevitably force programmers to write code in a certain way. Whether this is detrimental to coding remains to be seen. The third problem is that Java object serialisation is not an efficient method of making data persistent due to the fact that, like pickling, it is an all or nothing approach. There is no concept of saving only that data which has been modified since a particular point or time such as the start of a transaction. One can attempt to avoid this problem by selectively serialising objects. However, using such an approach it is easy to lose referential integrity which must be avoided. A final problem with serialisation is that it cannot be effectively used to save anything other than entire object closure. In many persistent systems the object closure may include the entire persistent store and perhaps even large portions of the Internet. If this approach is to be followed, techniques such as Farkas' OCTOPUS mechanism [4] or the use of weak pointers are also required.

The next approach to saving state is to provide persistence at the (Java) virtual machine level. This is the approach followed by Atkinson's PJava group [2]. Whilst we have argued elsewhere that the last approach is better, this approach has many merits in the application domain described in this paper. It also addresses many of the shortcomings of the serialisation approach described above. Providing persistence at a level lower than the Java language level permits the (reflective) type system to be broken and consequently all fields of objects may be saved to persistent storage rather than only the public ones. Secondly, the runtime state associated with threads may also be saved to persistent storage and later restored[†]. Since the runtime system has access to object implementations, it is easy to save only those objects that have been modified since a previous checkpoint or transaction start. This approach helps solve the closure problem described above although does not address the problem with respect to network transmission.

The final approach to providing persistence is to provide it at the virtual address space level; this approach is followed in the design and implementation of Grasshopper and has many desirable properties which we have described elsewhere [5]. One benefit of this approach is the ability to make all data in the address space persistent, including the state of threads and the stacks supporting them. Curiously, this does not assist in the transmission of state to another machine since data must be in an architecturally neutral format to support heterogeneity.

The above techniques are all capable of gathering some approximation to the persistent state of a computation. The format of the data in each case is different. Using the manual approach

[†] Current implementations do not support this functionality.

to saving persistent data produces persistent data in an ad-hoc format. This is a hindrance to its use in a general purpose system. Clearly some standard representation is required to enable the data to be saved and restored. In this respect, object serialisation is clearly the best approach. However, as described above, it is deficient in that it does not capture the dynamic state of computations. This is also true of the Aglet approach described below [6]. The most complete solution, that of persistence at the address space level, suffers from problems with heterogeneity, leaving the Pjava approach being the most promising.

The last problem is how to transport state between platforms and between platforms and servers. Once a format for the persistent data has been agreed, this may be easily achieved using one of the stream abstractions provided by Java that use TCP/IP.

4 Servers

Servers are responsible for four tasks in the architecture:

1. implementing the home of users,
2. providing persistent storage for non-persistent client platforms,
3. providing channel proxies, and
4. provide caches for client platforms.

The task of providing a home for users is the simplest of the four tasks. This requires the ability to record and recover the identity of the last hardware platform which implemented a view. As described above, this may be done simply and efficiently using existing Web tools and protocols. A server providing a user's home may also be required to provide persistent storage for (passive) data owned by a user. This is the traditional file/object server task often associated with the role of a server.

In addition to the file/object storage role, servers are required to provide persistent storage for the views implemented by the hardware platforms they support. This role is similar to that played by servers in support of early Sun diskless workstations (e.g. Sun 3/50). In this architecture the servers are likely to be required to support a cluster of diskless network computers. The state of the platforms is periodically checkpointed to the server which is responsible for saving the view on non volatile storage. This state may be requested by the platform following a crash or by another platform during the re-establishment of a view.

Servers also implement channel proxies discussed in Section 5.2. These provide a fixed location for communications with migratory hardware and software platforms. The fourth role of servers is that of cache manager. It is likely that clusters of network computers would often be running similar if not identical collections of code. Since servers support persistent storage and act as proxies for communications channels, it is natural for the servers to implement code and data caches.

5 Channel and Environment Mobility

5.1 Managing connection/reconnection

As described above, channel mobility may be implemented in two ways:

1. implementing software at both ends of the channel to manage the connection/reconnection, and,
2. using the server as a *connection proxy*.

In order to accommodate the connection and reconnection of channels we introduce a new abstraction called a *half session*. The primary purpose of a half session is to implement a communication channel which provides a reliable stream abstraction that can be disconnected and reconnected to different platforms and servers. As shown in Figure 4, on each platform or server implementing a relocatable channel, a half session is used to manage the connection. Thus there is a half session managing each end of a relocatable channel. The name, inspiration, and thinking behind half sessions is motivated by the seminal work of Strom and Yemini [7].

Half sessions present stream abstractions which are an extension of the interfaces presented by java.io.InputStream and java.io.OutputStream [8]. In addition to the methods provided by these interfaces, the half session abstraction provides methods for the re-establishment of the stream with an alternative client or server should a half session object be migrated. Clearly the re-establishment method must communicate with its peer half session in order to re-establish the channel.

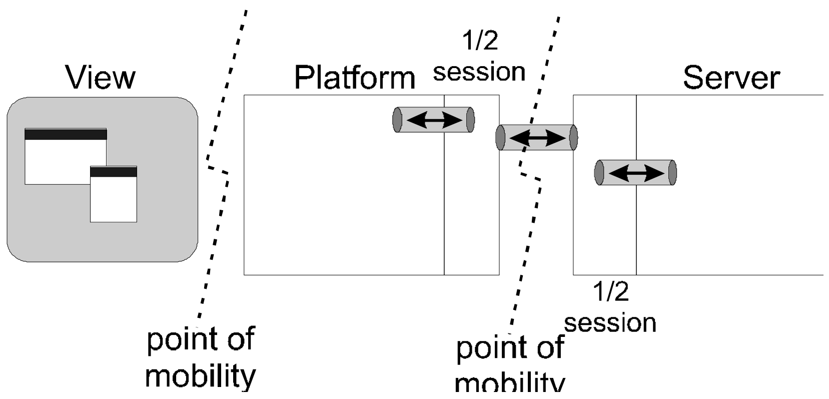

Figure 4: Half Sessions

5.2 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 which do not implement the channel abstraction. This is the case where a user is interacting with a legacy application running on a Unix system, for example a shell. Since legacy code does not support the half session abstraction, some additional mechanism must be provided to permit the platform (hardware or software or both) to migrate. This may be achieved by the use of the server as a fixed proxy for communication. Using this scheme, the fixed server communicates with the remote party on behalf of the platform. This communication is achieved using a traditional socket interface. The platform in turn communicates with the server using the half session abstraction permitting the platform to be relocated without the knowledge of the remote party which is only aware of the server.

The above scheme may be implemented in Java using an implementation of the java.net.Socket class which uses the half session objects described above rather than standard input and output streams. In the implementation of this class, all data is routed via the server using the half session abstractions rather than using direct communication with the remote party. This may all be achieved transparently to the client.

5.3 Agents and Channels

In addition to managing socket like streams, we also wish to support agent-style computation. This model is now well known [6, 9, 10] and requires autonomous computations capable of moving between different nodes in the network carrying code and data with them. Agents may be used to perform a number of tasks including scheduling meetings between different users and gathering information from a number of sites for example to arrange a trip. Consider this last example, a user may wish to travel between Stirling and California. Such a trip may involve agents visiting different sites containing information about train and flight times. After initiating agents to arrange a trip, the user may move to a different site. However, the agents should report back to the user not back to the site from which they originated.

The management of agents leaving from and returning to a view may be handled using a mechanism similar to channel proxies. All agents are routed via the server using the half session abstractions. This ensures that agents have a fixed location to which they may return. The server is responsible for holding agents attempting to return to views that are currently inactive.

5.4 **Binding to Services**

A final aspect of mobility that must be addressed is binding to the external environment. Binding to external services may either be dynamic or static, for example, an application running on a platform may wish to make use of a printer. In this case any postscript printer available locally may be suitable and the binding is dynamic. In other circumstances, for example when a user wishes to make use of a file/object server, only the file server containing the user's files would be appropriate. Here the binding between the application and the external service is static.

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 to the service is initially established. In order to support (re)binding activities, servers are required to provide an associative lookup mechanism like that provided by CORBA [11] and Grasshopper nameservers.

6 An Initial Experiment

As an initial experiment we have implemented an instance of this architecture to support a single application – a ubiquitous mailer. This system was constructed as a demonstrator and mimics the functionality of a ubiquitous mailer being implemented by DEC. The mailer is a Java applet which is uploaded from a server running the Grasshopper operating system. All persistent state is held on the Grasshopper system and loaded on demand to the mailer applet. The applet (800 lines) is essentially a mail viewer and contains code to authenticate the user, view mailboxes, compose mail messages and snapshot its state. Most mailer operations involve communication with the server. All interactions between the mailer and Grasshopper are made using HTTP. Since HTTP behaves in a connectionless manner, the mailer may move from platform to platform transparently to the server.

In the system described in this paper the snapshotting and recovery of platforms is automatic. In the mailer system, the state of the mailer is persistified (sic) by a thread with application specific knowledge. The thread sleeps on a timer and on awakening sends any volatile state of the mailer including messages currently being composed, lists of new mail etc. to the server using HTTP.

7 Related Work

7.1 Migratory Applications

In [9] an architecture designed to support migratory application in the language Visual Obliq is described [12]. Single user migratory applications are supported at the language environment level and may migrate from node to node whilst maintaining the state of their user interface. Almost no requirements are made of the application programmer which is the ethos behind orthogonal persistence and the thinking behind this paper. The basic building block of the system is the concept of 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 is migrated to the new site. When an agent arrives at a site, it is given a *briefing* which may include advice for the agent and site specific information. The above mechanisms have been used to construct migratory applications containing a *MigrateTo(Host)* command which causes the remote host to be contacted and if it will accept the application, it checkpoints the state of its user interface and performs a hop instruction. This work is complementary to our own and has coloured our own thinking.

7.2 Aglets

As described in [6] an *aglet* is a mobile Java object capable of visiting different hosts on a network. Aglets are autonomous, they each contain an active thread of execution and are capable of reacting to messages sent to them. 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. 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* which provides a uniform execution environment independent of the capabilities of the host. The aglet context serves to isolate the aglet from the platform.

Aglets have an *onCreation* method which is executed when the aglet is created or migrated to a new context. Aglets also contain a *dispatch* method which takes an URL as a parameter and may be used to migrate the aglet to a new context. When *dispatch* is called, the byte code and state of the aglet is preserved using standard Java object serialisation and transmitted across the network using the Aglet Transport Protocol (ATP) [6]. Aglets can be reactivated at the new site using the *onCreation* method.

Since aglets use Java object serialisation to export their state, the execution state of the threads owned by the aglet are not serialised. Therefore when an aglet is migrated or deactivated, any state resident on stacks and the program counters of running threads are lost. This is a consequence of the JVM, which does not permit direct access to run time state. Before an aglet is serialised, the host informs it that serialisation is immanent (via the *onDispatch* method) so that it may store any information it will need to continue its execution in object variables.

Aglets are complementary to the ideas in this paper in that they offer potential technology for implementing the architecture described in this paper. Whether or not the loss of dynamic state is too much of a programming restriction will remain to be seen.

8 Conclusions

This paper presents some initial thoughts on how persistent technology in general and Java in particular may be used to provide mobile users with a ubiquitous environment. A generally applicable architecture to support mobile users has been described. We have implemented a restricted prototype in order to validate these ideas. The architecture is realisable using technology which is currently available. Techniques with which user views may be located and restored on a different hardware platform have been described as have techniques for dealing with the difficult problems of inter-platform communication. These techniques both address the need to interact with other mobile computations and with legacy systems. They are also general and may be applied to mobile distributed applications unlike [9]. A number of engineering problems remain, in particular, the best way to save and restore closures of Java objects. We have suggested several approaches which may be used. The best approach will require further investigation.

9 References

1. Rosenberg, J. and L. Keedy, *Security and Persistence*. Workshops in Computing. 1990, Berlin: Springer-Verlag.
2. Atkinson, M.P., *et al. Design Issues for Persistent Java: a type safe, object oriented, orthogonally persistent system*. in *7th International Conference on Persistent Object Systems*. 1996: Springer-Verlag.
3. Javasoft, *Object Serialization Specification*.
4. Farkas, A. and A. Dearle. *Octopus: A Reflective Language Mechanism for Object Manipulation*. in *Proceedings of the Fourth International Workshop on Database Programming Languages*. 1994: Spinger_Verlag.
5. Dearle, A., *et al.*, *Grasshopper: An Orthogonally Persistent Operating System.* Computer Systems, 1994. **Summer**: p. 289-312.
6. Lange, D. and D. Chang, *Programming Mobile Agents in Java: A White Paper*, . 1996, IBM Corporation.
7. Strom, R. and S. Yemini, *Optimistic Recovery in Distributed Systems.* ACM Transactions on Computer Systems, 1985. **3**(3): p. 204-226.
8. Chan, P. and R. Lee, *The Java Class Libraries An Annotated Reference*. 1996, Reading, Massachusetts: Addison-Wesley.
9. Bharat, K. and L. Cardelli, *Migratory Applications*, 1996, DEC, Systems Research Center.
10. White, J.E., *Telescript Technoloogy: The Foundations of an Electronic Marketplace*, . 1994, General Magic Inc.
11. OMG, *The Common Object Request Broker: Architecture and Specification*, 1991, OMG.
12. Cardelli, L., *A Language with Distributed Scope.* Computing Systems, 1994. **8**(1): p. 27-59.