TCD Computer Science Technical Report
TCD-CS-2000-13

Title: Review of mobility systems

Authors: Tim Walsh, Paddy Nixon, Simon Dobson

Review of mobility systems

1 Motivation

Internet-distributed systems are beginning to offer a serious platform for stable, long-lived, flexible applications development. Moreover, such Internet-scale applications are motivating new modes of work and company integration such as teleworking and virtual enterprises. There is an increasing realisation that within such flexible working structures applications will have to support some form of mobility, both to handle explicitly mobile nodes (such as users with laptops) and to provide reconfiguration and redeployment of application components around the Internet. A major challenge for such mobile distributed applications is to provide stability in the face of constant change, providing application developers and users alike with a stable architecture whose elements and locations may change across time.

Virtual Enterprises, an association constructed from both administratively and geographically distributed business units or organisations, represent a new paradigm for doing business. At present e-commerce is limited to ordinary credit-card transactions. Instead, what is required is new set of tools that allow proper business to business transactions and contracts. Two separate events have made the virtual enterprise an attractive method of doing business. Firstly, the trend of companies to downsize and outsource. Secondly, the maturing of technologies such as CORBA and Java have made the concept viable.

Critical business applications require a quality of service which client-server invocations can not guarantee because of the unreliable underlying technology. The problem requires an inherent mobile solution that allows mobile objects the freedom of location transparency for predetermined, generic services. In effect the mobile object needs location transparency which is achieved by automatic rebinding services to the object. Examples include printers, displays and databases. Naming services will provide a standard naming structure for these services, allowing the system to service a visiting object's requirements.

Another domain of interest is the active or intelligent building. Active buildings promise to give greater access and usability to both able bodied and disabled people at work or at home. By performing tasks which should not need human intervention, people can concentrate on priority work. These 'Smart Environments' will perform planning tasks such as suggesting routes through the building to visitors, perception tasks such gesture and face recognition, action tasks such as calling lifts for robots, and software tasks such as managing the massive event reporting such real-world systems will generate. Facilitating this interaction requires the development of a variety of software infrastructures from support for actuators and sensors, through novel control and interaction interfaces, to distributed systems support. The software infrastructure behind such a system is again an intrinsically mobile one. Again location transparency allows computations which perform generic tasks to be sent to a computer, initialised and continue work.

2 Mobility

The term mobility is a very broad concept that can be broken down to three primary sections:

- Personal mobility

People do not necessarily want or need to carry around any physical piece of hardware. If a user goes to another location it should be possible for work to continue just as before regardless of the type or make of machine to which they are assigned. In many ways this idea is analogous to the situation of a person checking their web-based e-mail account. It is possible to carry on communicating with the exact same interface and setup regardless of platform or location in the world.

- Computer mobility

This refers to the issues involved in moving an actual piece of computer hardware from one place to the next. The proliferation of relatively cheap portable notebook PC's (on a par with their desktop equivalents) has essentially solved any problems with the mobility in terms of hardware. Also various pocket sized PDA's such as the 3Com palmpilot are a real alternative to carrying a relatively

cumbersome notebook computer. They have e-mail facilities, web browsers as well as standard desktop applications.

- Computational mobility
  This is the area of interest to us. It encompasses many different forms that are covered in section 2.1. The basic idea involves migrating the source code to another address space, typically on a physically remote node. Once it has successfully reached its destination it is (re)started depending on the type of migration employed.

2.1 Generic Mobility Issues in Computational mobility

The notion of mobility has acquired prominence in a variety of forms. Cardelli [1] lists the following as the basic forms of mobility:

- Control Mobility: Occurs during an RPC or RMI call where the thread of control is thought of as having passed from one machine to another and back again.
- Data Mobility: During an RPC or RMI call the method name, its arguments and return type are serialised, packed and transmitted to and from the remote machine
- Link Mobility: The end-points of network channels, or remote object proxies, can be transmitted.
- Object Mobility: Objects moving between nodes allowing work to be performed on-site and to assist in load balancing
- Remote Execution/Evaluation: Computations can be shipped for execution on a server.

Mobile computation [2] is the notion that a computation starting at some network node may continue execution at some other node. It encompasses most of the points in the previous list especially if working in an object oriented environment. In a mobile computation the control must be transferred by migrating to a new host and continuing execution. There are three distinct models for achieving this:

- Basic mobility: Consists of moving just code. The state of the computation is lost at the originating node. State and connectivity (if any) must be re-established at the destination node.
- Mobile Computation [2]: The state of the execution is migrated with the code so that the computation can be (re)started at the receiving end. The amount of detail captured in the state asserts the type of mobility employed, either weak or strong (see below). In the object oriented paradigm these mobile computations have been commonly referred to as autonomous objects. These objects can retain references to other nodes and objects across the network.
- Mobile Agents: In the mobile agent paradigm [5] the agent migrates to a foreign node, carrying computational ‘know-how’. This is similar to autonomous objects but agents should not retain references to other entities outside the host node. Instead they use resources made available to it on the host node. It is generally accepted that there are two broad types of agents: autonomous agents and intelligent agents. The former typically have a set of tasks to perform and an itinerary to follow. These have no inherent artificial intelligent The latter are the subject of research of the AI community [31] and use different techniques to exhibit self-determining traits.

The transfer [5] of code and state enacts data mobility while network references to, for example, other objects account for link mobility (at least in models which support remote connections).

We are interested in mobile objects. The concept is simple and elegant; an object (that may be active or passive) that resides on one node is migrated to another node where execution is continued. The [3] main advantages of mobile computation, be they object based or not, are as follows:

- Load balancing: Distributing computations among many processors as opposed to one processor gives faster performance for tasks which can be fragmented.

- Communication performance: Active objects which interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction

- Availability: Objects can be moved to different nodes to improve the service and provide better failure coverage or to mitigate against lost or broken connections.

- Reconfiguration: Migrating objects permit continued service during upgrade or node failure.

- Location Independence: An object visiting a node can rebind to generic services without needing to specifically locate them. They can also move to take advantage of services or capabilities of particular nodes.

Karnik and Tripathi [4] also point to the increased asynchrony between client and servers. The client need not maintain a network connection with the server. Instead the client agents run on the server while the client performs other tasks. This is particularly useful when poor or unreliable connections exist between the two machines. Likewise a mobile object residing on a server can add higher level functionality (or specialised interfaces) using a fixed set of server supplied primitives. This removes the need to change the server interface to meet the requirements of one specialised use.

Many Mobile Object (MO) Systems exist and although their underlying principle is the same, namely to migrate an object to a foreign host, the mobility mechanism they use to achieve this varies considerably.

Figure 1 from [5] presents the taxonomy of MO systems in general. The hosts represent the physical machine on which the code executes and the core operating system, such as NT or Solaris provides access to hardware services. The network operating system represents a TCP/IP stack. The computational environment (CE) represents the mobile object system and the components denote mobile objects. The component comprises of a code segment, an execution state (stack and instruction pointer) and a data space. The CE can send a component to another node where it's CE will restart the component. A typical example of such a system is a Java applet embedded into a web page. The Java bytecodes represent the component and the computational environment is implemented in a Java virtual machine that the web browser starts.

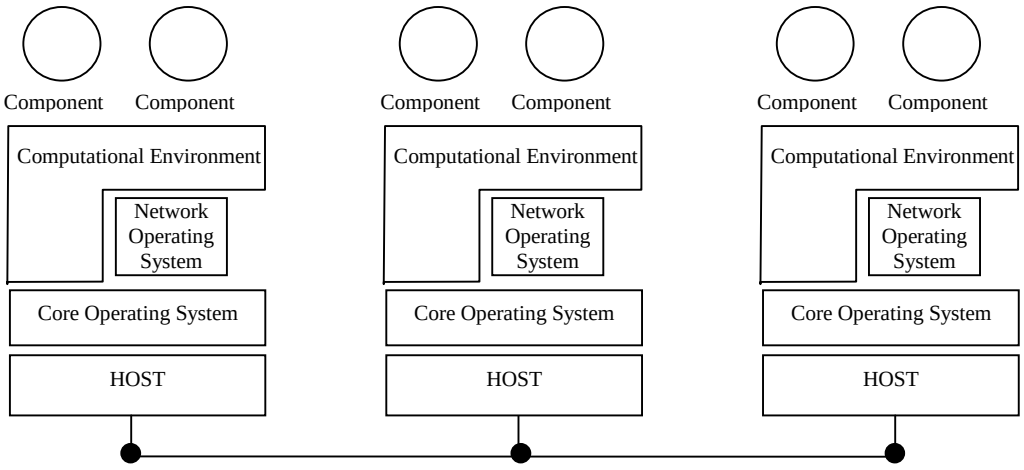

Figure 1: Mobile Code Systems [5]

The manner in which component are migrated depends on the manner in which the CE is implemented. As we a mainly interested in objest oriented systems the term mobile object will take the place of component. The mobile object is essentially a piece of compiled source code. What makes it unique, from the many possible instantiations that may exist of it, is its state. Therefore, for any object migration to be useful it is essential that state is also migrated. Fuggetta et al [5] define strong and weak migration as the two distinct types of migration.

Modern [25] mobile agent systems which use the standard Java VM's typically transfer the instance data of objects from the source to the destination and restart execution on the remote platform. Mobility system which do not provide transparent mobility tend to be more complicated to program. Compare the Figure 2 and Figure 3 below.

```
public class Test extends Aglet {
      boolean Remote = false;
      public void onCreation (Object init) {
            addMobilityAdapter() {
                  new MobilityAdapter() {
                        public void onArrival (MobilityEvent e) {
                              Remote = true;
                        }
                  }
            }
      }
      public void run() {
            if (! Remote) {
                  System.ouit.println ("On Source");
                  dispatch (destination);
            }
            else {
                  System.out.println("On destination");
            }
      }
}
```

Figure 2: Mobile object using weak mobilty

```
public class Example extends Agent
{
      public static void main (String[] argv)
      {
            System.out.println ("On source");
            go (destination); // object migrated
            System.out.println ("On destination");
      }
}
```

Figure 3: Mobile object using strong mobililty

Fuggetta [5] defines **strong mobility** as the ability of a CE to migrate the code and execution state of the component. This includes program counter, saved processor registers, return addresses and local variables. The component is suspended, marshalled, transmitted, unmarshalled and then restarted at the destination node without loss of data or execution state. Examples include Emerald, which was one of the first systems to implement this in an object-oriented environment. (See section 3.1) Systems that implement strong mobility are usually proprietary in nature.

Weak mobility is the ability to allow code transfer across nodes; code may be accompanied by some initialisation data, but no migration of execution state is involved.

It can be argued that in between these two extremes a different form of migration takes place. It is a *pseudo*-strong mobility as a partial snapshot of the state is taken (see section 2.1.1). While the program counter and local variables are not preserved, the global variables are. If no methods are being executed when migration is initiated then the stack will be flat (no local variables, program counters or registers saved), thus allowing the object to be moved and restarted on a new node. It should be noted that the term 'moved' is somewhat misleading. In practice the object is actually being cloned and transmitted. The original object needs to be destroyed for the illusion of a 'move' to be complete. Undiscriminate removal however, may be undesirable in a fault tolerant environment.

Another task needed in the migration of objects is to maintain the bindings the object has to other objects. The data space holds the set of bindings to resources accessible to the mobile object. Data

space [5] management is the process of voiding certain binding, re-establishing new bindings, or even migrating some objects to the destination MO system along with the object.

2.1.1 Java as an example

In terms of weak mobility Java can provide a good example of Fuggetta's [5] weak mobility systems. In Java's case, an object is being transferred as code and if required, initialisation data,. No execution state is migrated. Java's applets are quintessential weak mobility. The bytecodes have initialisation data when migrated from the web server they have not been suspended so have no state.

As already mentioned, strong mobility systems tend to be proprietary in nature. Java is a non-proprietary, mainstream language. Its serialisation mechanism does not capture state at the thread level. Instead, it serialises [6] all non-transient and non-static data members. Although threads, strictly speaking, are Java objects they are not capable of being serialised. For Java to migrate transparently, thus satisfying the strong mobility definition, requires rewriting of the virtual machine which can make it incompatible with existing VM's. Another [28] technique is to insert code that captures local variables at runtime. This requires a pre-processor and introduces undesirable overhead.

Alternatively, Java offers a mechanism allowing a type of object migration that lies between the strict definitions of strong and weak laid down by Fuggetta[5]. Using object serialisation the virtual machine gives access to the object's member variables (except static and transient members) which allow the programmer to send them to another host where they are deserialised. If the bytecodes do not exist at the receiving end then a specialised classloader allows them to be downloaded from any available source. With the object's state and bytecodes on the same virtual machine the object can be re-started. Java does not provide a mechanism to do this. Instead, using the reflection API, the programmer can access a pre-determined method name and call it. This method allows the object the chance to perform housekeeping restart execution.

The Java externalisation interface gives more control over what is serialised but even though it is possible to get some information on a thread (`getName()` and `getPriority()`) no mechanism is provided to check point the code. Hence, this approach does not allow for the object to be stopped mid-method, as is the case with Emerald and Agent TCL. However, it can be argued [6] that if each method invocation is treated as a single unit of execution then serialisation can perform strong migration. That is, after a method is finished executing, its entire state is quantified in its member variables so serialising it encapsulates its full state.

The remainder of this paper examines various mobility systems, both research and commercial, available. Some implement strong mobility, others weak mobility. They all exhibit attractive features but the primary purpose of this paper is to demonstrate the fact that there is no system is capable of fully satisfying the requirements of a Virtual enterprise or intelligent building. Namely the locations transparency achieved by automatic rebinding of services to an object. We propose a system that specifically addresses the issues in these domains. Due to its inclination to the Internet, Java will be the development language.

3 Mobile Object Systems

3.1 Emerald

3.1.1 Overview

Emerald was a project undertaken in the mid 80's at the University of Washington. Although it is no longer state of the art, it was one of the first functional object based mobility systems and represented a new paradigm for migrating objects. Its aim was the construction of an object-based language for distributed programs. An explicit goal was to support fine-grained mobility but retain a uniform object model. Emerald objects could range from small passive data objects to active process objects. It implemented strong mobility with the specific purpose of running on a relatively small (<100) number of nodes. Its deployment on larger, long haul networks, such as the Internet, was never planned nor designed for.

The designers saw the importance of a single object model for simplicity but also the need for distinct implementations for when objects are distributed or local. To allow for this the compiler analyses the needs of each object and generates an appropriate implementation. The compiler produces different implementations from the same source code. Direct memory addresses in Emerald increase performance of local invocations but also means moving an object requires finding and modifying all direct addresses. This places the price of mobility on those who use it.

3.1.2 System

The objects which populate an Emerald system have at least three components which contain all necessary information; unique network-wide name, data local to the object (primitive data and pointers to other objects), and set of invokable methods. It can also contain a process depending on whether the object is active or passive. Monitors provide synchronisation for when multiple threads of control are active concurrently.

Emerald supported the 'abstract type' which is now commonplace in modern object oriented languages. An object [5] *conforms* to an abstract type if it implements as least the operations of that abstract type and if the abstract type of the parameters conform in the proper way. An abstract type [7] is defined by a collection of method signatures, each of which describes one method – its name, parameters and return types. Unlike modern programming languages, such as Java and C++, Emerald has no similar class/instance hierarchy. Objects are not members of a class; conceptually each object carries its own code but in practice identical objects on the same node share the code. In the implementation, the code is stored in immutable *concrete* type objects that are not automatically sent when an object migrates. Instead, only the object's data is moved. The destination kernel determines if it requires a copy of the concrete object. Concrete objects remain on a node for as long as there are objects referencing them, after which they are garbage collected.

The mobility mechanism is provided by a small set of language primitives:

- *Locate X:* returns the node where object X resides.
- *Move X to Y:* moves object X to node Y
- *Fix X at Y:* Prevents object X from moving from node Y
- *Unfix X:* Makes object X mobile again
- *Refix X at Z:* Atomically performs an Unfix, Move and Fix for node Z
- *Attach* semantics are provided to the programmer allowing for specific objects to be attached to(and hence moved with) other objects. An example of this would be the attaching of an array to an object that implements a queue algorithm. Should the queue object be moved without the storage array then the subsequent remove access will cost the system severely at run-time.

Based upon knowledge of an object's use, the compiler chooses between three different object implementations. *Global* objects move independently of each other, have a globally unique name and can be invoked by objects not known at compile time. *Local* objects are conceptually contained within another object. When the senior object moves the local object is automatically moved. For example if an object creates an array the compiler would class the array as a local object and attach it to the original object. Local objects are accessed using local procedural calls and are heap allocated. Finally *direct* objects are used for primitive types, structures of primitive types and other simple types whose organisations can be deduced at compile time.

As mentioned previously, global objects have a globally unique name. The local reference to a global object points to the object's *descriptor* that contains information about the state and location of the global object, including a flag indicating if the object is resident. If it is resident the object's descriptor points to the Object's Data area otherwise it contains a forwarding address. The forwarding address is the last known location of the object and is not necessarily where the object now resides. Nodes keep hashtables mapping object identifiers to the object descriptors. Entries are held for local objects that have remote references and remote objects that have local references. Nodes that no longer host an

object forward requests to the node the object moved to. The moved object will then reply to the object that originally requested it.

All objects which have method invocations have an activation record representing the part of a stack applicable to that object. As a process can span a number of objects the process stack is visualised as per Figure 4, where the process owned by object A invokes operations in objects A, B and C.

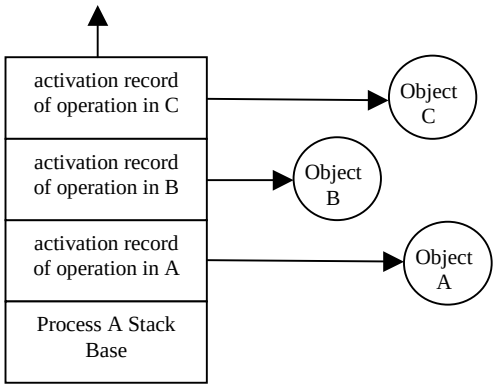

Figure 4: Process Stack and Activation Records [3]

When an object is moved to another node the activation records for any threads which are invoking functions in that object are also moved. Figure 5 demonstrates this for one thread of execution. When the method in object C returns, control passes to Node X where the method in object B continues executing. After completion, control is returned to Node Y where the process continues executing. Once again, process A is owned by object A.

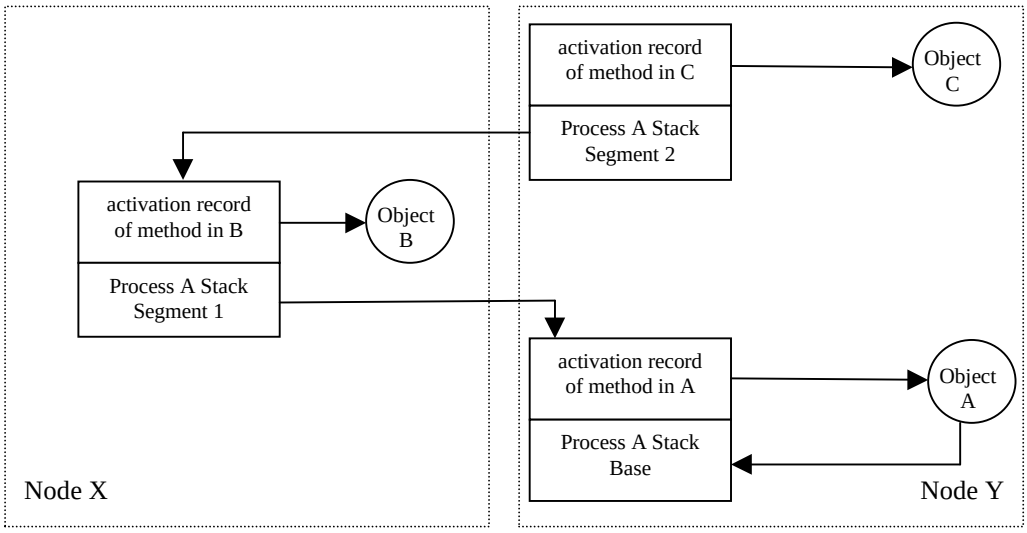

Figure 5: Process stack after object move [3]

The moving of activation records is necessary for the system to implement strong mobility. The executing process will follow where its objects go, thus insuring data integrity. This creates a large overhead that would not be practical on a large number of nodes as it leaves the system very vulnerable to failure.

3.1.3 Summary

The Emerald system is a proprietary object oriented language that implemented strong mobility for the construction of distributed programs. Its fine-grained mobility model allows the migration of small data

objects to process objects. It was designed for small clusters of machine (about 100 nodes) and due to its design was unsuitable for large unreliable distributed system.

3.2 FarGo

3.2.1 Overview

FarGo is a weak mobility system designed in Java at the Israel Institute of Technology. It seeks to separate the programming of application logic from the layout of a program since a designer is unlikely to know a priori how to structure an application in a way that best leverages the available infrastructure. The developers [8] argue that a dynamic application layout elevates system scalability. The implicit and explicit approach (see section 3.2.2) to distributed programming hamper either the programming scalability or the system scalability. According to the developers of FarGo dynamic layout, where local-remote relationships and the overall mapping of the application logic onto a set of physical hosts is manipulated at run-time, does not display such adverse effects.

3.2.2 System

The FarGo designers weighed up the defferences between implicit and explicit programming. The advantages of taking the implicit approach to distributed programming appear to be a perfect remedy to ease the construction of distributed applications. A unified object model makes remote method invocation syntactically and semantically equivalent to local method invocation.. Programming scalability refers to the ease at which the distributed system can be expanded. If a programmer can use objects without regard for their location it makes the task of building the system far more straightforward . Simple normal object invocation allows the programmer to concentrate on the program functionality. The unified object model takes care of finding the objects and marshalling all data.

However, as Waldo [11] points out, objects that interact in a distributed system are intrinsically different from objects interacting within a single address space. The programmer must be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure. Such issues do not arise on a single node. Therefore, the implicit approach does not support system scalability. The larger the system becomes, the more numerous the delays and the greater the chance of failure. The semantic differences between local and remote arise because the local model assumes there is shared memory, allowing pass-by-reference argument passing. Distributed Shared Memory (DSM) could be used to eliminate such a dilemma but as studies [15] have shown, DSM is not practical on a global scale. DSM is primarily of use on a cluster or processor cubes.

The explicit approach toward distributed programming dictates that the programmer *explicitly* distinguishes between local and remote objects thus allowing system scalability. This tight coupling between the application's logic and distributed programming has a negative effect on the programming scalability. Technologies such as CORBA [12], RMI[13] and DCOM [14] let the programmer use distributed objects as if they were local. The distributed objects are fixed and the programmer typically knows their location. Accepting [8] this model requires layout of the system's architecture, including local-remote interactions, to be set down at design-time. This still impacts on the system scalability but not to the same extent as the implicit approach.

The FarGo designers [8] favour the dynamic layout of a system. Their model specifies that local-remote partitioning and overall mapping of application logic onto physical nodes is manipulated at run-time without any change to the source code. This [10] better reflects dynamically changing computing and network resources since it is impossible to predict these at the coding stage.

The two basic requirements for such dynamic layout support are identified as

- Code mobility - the ability of actual code and state to move from one host node to another
- Tracking facility – The ability of separate components to find and interact with each other regardless of movement.

The following further prerequisites are what allow for FarGo's unique system operation:

- A mechanism which makes decisions regarding component relocation. Decisions are based upon quality and availability of resources.
- A means of specifying binding types [5]. These specify how mobility can affect other components which have references to a migrating component
- Preserve as much of the local programming model as possible. For inherently remote actions such as parameter passing across remote invocations this is can not be possible.

FarGo's system is implemented using the complet and core abstractions. The core is the static sandbox for complets to execute within. Complets are the basic building blocks of a FarGo application. They consist of a collection of local objects, i.e. they share an address space and all references among these objects are regular local references. Complets are the minimal unit of relocation. Each complet has a single *anchor* object that represents the interface of the complet. Given the anchor [9] a of a complet α, the set of objects that comprise the complet, termed the *closure(α)*, is defined recursively as follows:

- $a \in$ closure(α)
- If $b \in$ closure(α) then $\forall$ c such that b references c, if $c \neq$ anchor (β), ($\beta \neq \alpha$), then $c \in$ closure (α)

Therefore, objects within a complet can only reference each other or reference other complet's anchors (referred to as complet references, see Figure 6). Objects within a complet can not reference other non-anchor objects in other complets.

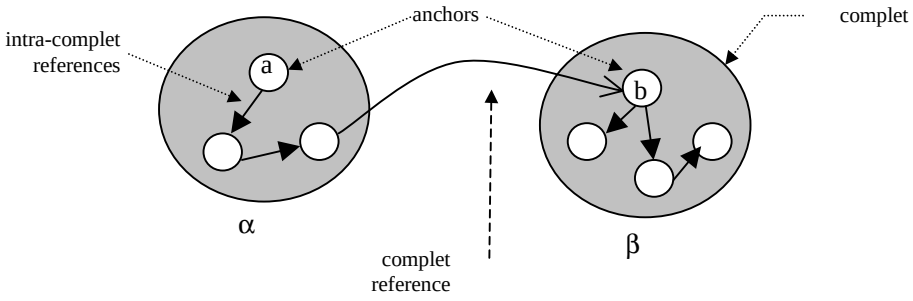

Figure 6: Complet Structure [9]

This clean distinction between inter-complet references and complet references allows these references to be treated differently at runtime. Intra-complet references are always local, allowing fast pass-by-reference operation. Complet references are a vital abstraction for migration semantics. Across a complet reference regular (non-anchor) objects are passed by value, while anchor objects are passed by complet reference.

Complet references are the abstraction used enabling components to communicate, hence allowing applications to be built from various component parts. They remain valid regardless of a complet's migrating tendencies. This is accomplished by using a stub that references a tracker object. Any node where the complet is referenced retains a copy of this stub and tracker. The tracker provides the location transparency. The stub has the same constructor interface as the anchor it references. A residual tracker object is left on each core that the complet visits. The tracker forwards messages to the complet's tracker on the destination site. This may point to another residual tracker should the complet have since migrated once again. The runtime shortens these 'follow-me' chains at pre-determined time slots or after every method invocation. See Figure 7.

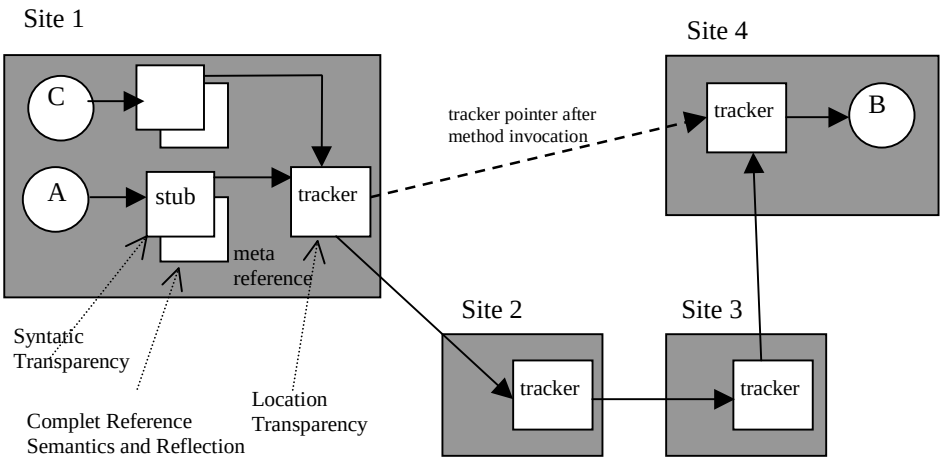

Figure 7: Complet Reference Implementation[8]

The complet's reference type [9], which can evolve dynamically, also provide a means to control the layout of the application by associating with it, co-location and re-location semantics. When α complet's references β complet, α is referred to as the source and β is referred to as the target of the reference. The five basic type of complet reference are:

- Link: This is the most basic reference type. This link remains valid regardless of each of the complet's migration patterns.
- Pull: This reference ensures co-location. When the source of the reference moves, it takes the target complet with it, thus complying with the co-location requirement. A target complet can only accommodate one Pull reference. Otherwise, if two separate source complets (A and B) referenced the same target complet (C) then a migration of A pulls C. B is also pull referenced to C but C is no longer in the same Core as B which contradicts the co-location rule. Targets can not move their source. Such references are of value if they communicated frequently or require heavy data-transfer.
- Duplicate: Both source and target are co-located. Migration of the source takes a copy of the target with it thus staying consistent with its co-location obligation. This reference is particularly useful when the target is a small immutable or read-only object.
- Stamp: A stamp reference allows a complet to reference a particular type of object that is typically available at a core. Examples include services such as displays and printers. When a complet migrates to a host it seeks to rebind its stamp reference to a local instance of the original target's type.
- Bi-directional pull: This reference is similar to the Pull reference except that bi-directional pull allows either target or source to migrate and pull the opposite complet with it. This reference is effective in defining group re-location semantics. Relocation of any of the group members leads to relocation of all members of the group. Like the Pull reference B can not be referenced by another Pull reference. This resolves the possible conflict of having both parties being pulled to different locations, e.g. A to node X and B to node Y.

Since execution state is not captured at the thread level (weak mobility) FarGo requires a continuation mechanism to allow the complet to continue after migration has taken place. Two such methods are provided.

- A 'call with continuation' method similar to Voyager [16] or Migrants [17]. An invoked method which moves the complet is supplied with destination, restart method and parameters for this restart method.
- A callback mechanism is supplied by the programmer and is invoked by the movement protocol in different phases of the movement. The methods are pre- and post-Departure which are invoked on the source machine, and pre- and post-Arrival methods executed before and after arrival of the complet(s) on the destination machine.

These allow the complet to have restarting abilities that Java does not specifically supply in the Virtual Machine specification.

Although the complet references allow the programmer to structure the application, other services are available which control relocation and so implement a dynamic layout structure. A profiling and event service is available to allow an administrator to specify relocation semantics when certain conditions prevail. The profiling service give details of system conditions such as the number of complets in the core and the bandwidth between hosts and thus allows policies to be scripted which manage the complets and cores. Also associated with the profiling are events which complets can subscribe to. This gives asynchronous control. For example, a complet relocation fires a *completDeparted* event which a different complet could subscribe to with the intention of killing itself once that particular complet departed.

3.2.3 Summary

FarGo implements a weak mobility type system, the abstract type of which is described in [5]. Its primary goal is separate the programming layout from the programming logic through the use of dynamic layout. This capability introduces high benefits for both programming scalability and system scalability of distributed applications. The explicit or implicit approach has negative effects on programming and system scalability respectively. The binding type abstraction provides a data space management system, which the FarGo Core administers.

3.3 Aglets

3.3.1 Overview

Aglets were created by IBM's Tokyo research lab. Aglets are full Java objects that can move from one host on the Internet to another. The Java Aglets API (JAAPI) system [18] is essentially a weak mobility agent based system because it supports the ideas of autonomous execution and dynamic routing on its itinerary. Along its design goals were: Simplicity and extensibility – the API would be simple to use, Platform independence – portable to any platform running the JAAPI, Industry Standard – that JAAPI would become the industry standard, Safe – untrusted Aglets should not be considered a security risk for the agent host.

3.3.2 System

The Aglet object model [18] defines a set of abstractions and the behaviour needed to leverage mobile agent technology in Internet-like open wide-area networks. Figure 8 shows the form of the Aglet system model.

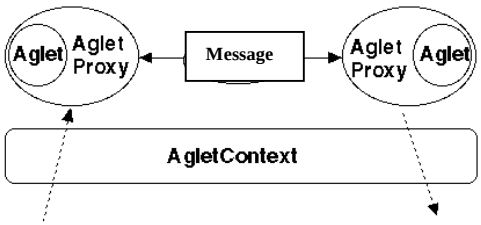

Figure 8: Aglet System Model

The AgletContext (henceforth referred to as the context) constitutes the computational environment or sandbox within which the Aglets run. A single node may play host to several contexts. It is implemented as a stationary object that provides a means for maintaining and managing running Aglets in a uniform execution environment where the host system is secured against malicious Aglets. The context provides a set of basic services, e.g., retrieval of the list of Aglets currently contained in that context or creation of new Aglets within the context.
The Aglet is a mobile Java object that visits Aglet-enabled hosts in a computer network. It is autonomous, since it runs in its own thread of execution after arriving at a host, and reactive, because

of its ability to respond to incoming messages. For security reasons the Aglet is shielded by a proxy interface. This shields [19] potentially malicious Aglets from directly accessing its public methods. The proxy also provides location transparency for an Aglet. If the actual Aglet [32] resides at a remote host, it forwards the request to the remote host and returns the result to the local host.

The Aglet class provides the basic functionality for the mobile object from which the programmer subclasses and specialises. Creating an Aglet requires either the cloning of an existing Aglet (which receives a new unique AgletID) or requesting the context to create a new instance of one. Once created [19] , an Aglet object can be dispatched to, and/or retracted from (see below) a remote server. They may be deactivated, placed in secondary storage, then activated later. An *itinerary* provides the abstraction for the travel plan of the Aglet. Since an Aglet object is active it can decide when to die. A *disposal* method is supplied to halt execution and remove it from its context. However, the Aglet programmer is responsible for releasing resources such as file descriptors, thus allowing the object to be garbage collected by Java.

Aglets communicate with each other by passing Messages (specifically objects of type Message). Message passing, which may be synchronous and asynchronous, can be used by Aglets to collaborate and exchange information in a loosely coupled fashion.

Migration primitives are kept to a minimum; *dispatch* immediately and asynchronously sends the Aglet to a destination context, *retract* synchronously forces the Aglet to return to the calling context. As mentioned previously, Aglets are a weak mobility system since Java, the underlying technology, does not provide for stack frames to be captured or thread objects to be serialised. Instead an event system is supplied which can perform house cleaning if, for example, a migrating Aglet needs to close windows and file handles. This allows the stack frame to be reduced to nothing. The global members are serialised and sent to the destination node. Upon arrival an event can be generated which sets up resources (see Figure 9). To an extent, this mimics the action of a migrating process. Other events include: Clone Event and Persistency Event, which is generated when an Aglet is placed into secondary storage, for instance, on a hard disk.

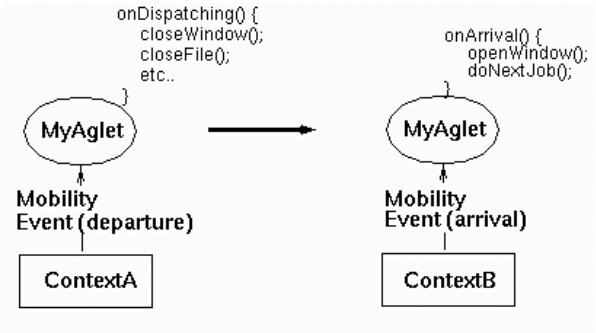

Figure 9: Aglet Event Model

Aglets also provide a callback mechanism [20] for when a primary action (creation, destruction, migration) is performed on an Aglet. These methods invoke a corresponding method which allow the Aglet to have the final decision. This mechanism prevents other malicious Aglets from forcibly destroying or migrating an Aglet.

Cardelli [2] states that agents are meant to be completely self-contained. They do not communicate remotely with other agents, rather they move to some location and communicate locally. Aglets are classified[18], by their developers, as agents and as such do not have the binding type abstraction mechanism as seen in FarGo (section 3.2).

The Aglet [19] architecture consists of two layers (Runtime and Communication Layers) and two API's (Aglet and Communication API) which define interfaces for accessing their functions.

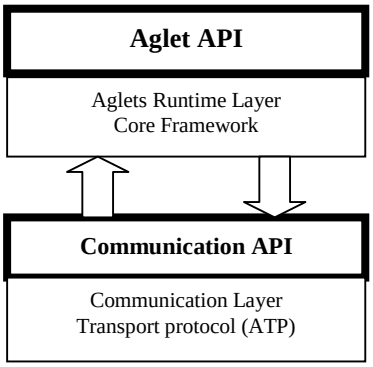

Figure 10: Aglet Architecture

The Aglet Runtime layer implements the Aglet API and defines the behaviour of API components, including the *AgletProxy* and *AgletContext*. It provides the fundamental functionality for Aglets to be created, managed and dispatched.

The Core framework provides the functions essential to Aglet execution. That is:

- Serialisation/Deserialisation of Aglets which are then handled by the communication layer
- Classloading and transfer. Required when local copies of the class's bytecodes are not present on the destination node.
- Reference management and garbage collection.

The communications layer administers sending and receiving serialised data to and from other contexts and Aglet applications. The protocol used to transmit the data is interchangeable. The Aglets Transfer Protocol (ATP), a pseudo-HTTP, is the default but other options include RMI and there are plans for a CORBA ORB.

The security of Aglets is not, due to the limited encryption support in the JDK, enterprise strength but the creators consider it to provide a *reasonable* level. The system endeavours to give protection from hostile agents. It provides mechanisms to authenticate users and domains, and integrity check inter-server communication. Aglet servers running in the same domain share an authentication code. This authenticates between servers that can now exchange Aglets. The receiving server has to decide how much to trust the Aglet depending on the credentials it supplies. Integrity checking is achieved with a different integrity code. Agents from same domain by default are accepted upon their credentials but the server may choose to downgrade an agents authenticity or deny access depending on (amongst others) where the agent has been roaming.

3.3.3 Summary

Aglets implement a weak mobility framework in a manner that is familiar to many other systems in its class. It has a simple API that allows the programmer to write mobile objects in an intuitive manner. All Aglets have proxies as their means of access that protects them from other malicious Aglets and thus provides a reasonable level of security. The proxy also provides location transparency for the Aglet. Inter-Aglet communication is message based. Aglet's event model allows the programmer to perform tasks when key events (such as onDispatching()) occur.

3.4 Voyager

3.4.1 Overview

Voyager [6] is a Java agent-enhanced ORB (Object Request Broker) system. This means that although Voyager has a subset of the capabilities found in most ORB products, including support for the CORBA standard, it also includes a number of features that would suit an agent toolkit. Voyager was built with mobile agents and distributed objects simultaneously in mind. Voyager implements weak mobility.

Rather than supporting a single standard for multiple languages, Voyager supports multiple standards for a single language, namely Java. It also simultaneously supports all the major distributed system paradigms, namely CORBA, RMI and DCOM. It can dynamically generate proxies which removes the need for stub generators.

The basic package includes a universal naming service, universal directory, activation framework, publish-subscribe, and mobile agent technology. The professional edition provides enhanced facilities like Java Naming and Directory Interface (JNDI) integration and CORBA naming services. There are other Voyagers modules that provide security, distributed transactions and enterprise JavaBean development environments.

3.4.2 System

Voyager is primarily a mechanism for the creation of distributed applications through the use of its core ORB. Typically, in ORB based systems, the programmer wishing to deploy remote objects needs to firstly define the object's interface. The OMG's CORBA specification for objects was intended to allow communication across different computer platforms and different programming languages. This functionality has side effects on ease of use. Therefore the interfaces need to be defined in a neutral fashion. The Interface Definition Language (IDL) provides such a system.

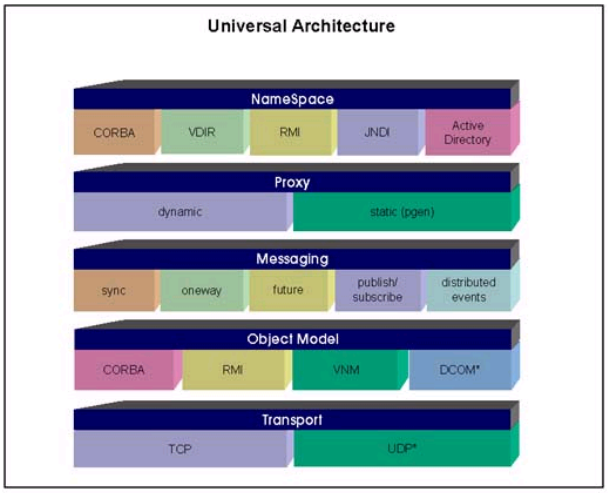

Figure 11: Voyager Architecture [16]

The IDL compiler generates language specific code (e.g. Java) from IDL definitions. This includes client stub code, for development of client programs, and server skeleton code, for the server objects. When a client calls a member method in the remote object, the call is intercepted and transferred through the client stub to the ORB. The ORB then passes the method call through the server skeleton code to the target object (see

Figure 12). The ORB then collects results from the method call and returns these to the client. The stubs and skeleton are often referred to as proxies since they act as 'go-betweens' for the programmer. Later versions of Voyager provide a mechanism that dynamically creates the client stubs and server skeleton proxies.

All ORBs on the market, including IONA's [21] Orbix and OrbixWeb, essentially provide the same basic service. The ORB allows the programmer to create standard software objects whose methods can be invoked by client programs located anywhere on the network. The Voyager remote objects are referred to by two different methods. Either a URL type name is specified in the form *node:port/Objectname* or a globally unique ID (GUID) is specified.

Voyager [22] has the ability to turn any Java class into a remote enabled class, regardless whether the source is available or not. Version 1 of Voyager had a Virtual Class compiler (vcc) utility that created 'virtual' classes. This virtual class handled all communication with the actual class running on a Voyager server. Later versions removed this facility allowing for automatic stub generation. There is no need for an IDL compiler (unless specifically requested by the programmer). The programmer simply creates the interface, extends *IRemote* and implements the interface (which requires no special code or modification). Both the client and server start a Voyager process that takes care of all remote enabling.

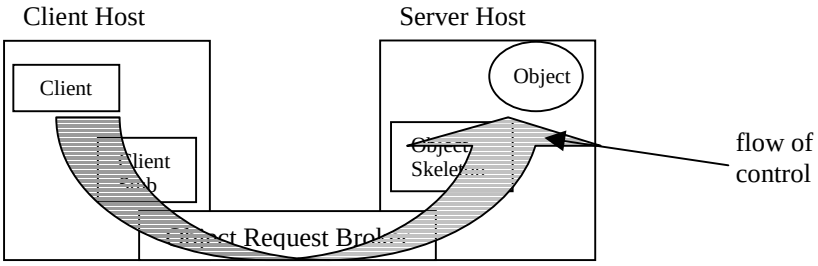

Figure 12: Object Request Broker

Using Voyager for this client-sever interaction requires the Voyager runtime to be started. The runtime can then also be used as an agent server, a platform for hosting mobile objects, essentially the sandbox for Voyager. The Voyager process can also be run as a standalone daemon.

Voyager [22] uses standard object serialisation and sockets for communication, but one can easily specify that a server should use another Socket Factory (e.g. secure sockets). Furthermore IIOP is supported as transport layer. The standard form of messaging between object is synchronous with no time out. There are however several possibilities, such as Future (asynchronous), OneWay (no return value is required), OneWayMulticast (sending a one way message to a group of objects or to objects who satisfy a certainly criteria) and finally the default Sync (synchronous).

There is a single migration primitive *moveTo()* which the programmer uses to specify a destination host or destination object, a call-back method used to restart the object, and the optional call-back method's arguments. Like FarGo (section 3.2) there is also a mechanism provided to obtain move notification. These are the *preDeparture()* and *postDeparture()* methods which are executed on the original source node and the *preArrival()* and *postArrival()* methods which are invoked on the destination node.

Voyager was designed primarily as an ORB. Its agent technology is simple and basic but does not provide anything that differentiates it significantly from Aglets.

3.4.3 Summary

Voyager, amongst its many features, is a weak mobility system for mobile objects (Voyager's documentation refers to them as mobile agents). The migration primitives are similar to Aglets such that the *moveTo()* method takes a URL location. However it also takes string call-back method name which is automatically called by the destination voyager daemon. Also, unlike Aglets, there is no event system to catch key events.

3.5 Agent TCL

3.5.1 Overview

Agent TCL (tool command language) is mobile-agent system created at Dartmouth College. It was created primarily for mobile devices (laptops, palmtops) which allow for disconnected operation. Migrating agents, which are typically TCL scripts, implement strong mobility, capturing execution state at the thread level. Agents have location dependent identifiers based on DNS hostnames, which therefore change on migration. The system provides safety mechanisms that prevent the host from malicious agents although not the other way around.

3.5.2 System

The architecture of the Agent TCL system is illustrated in Figure 13. Agents come in two guises; mobile and stationary. Mobile agents are free to relocate other hosts. Stationary agents remain in place. The only difference between them, from the perspective of the system, is that stationary agents have more access to system resources. Stationary agents tend to supply services such as network monitor, navigation, high-level inter-agent communication, docking (see later) and resource management.

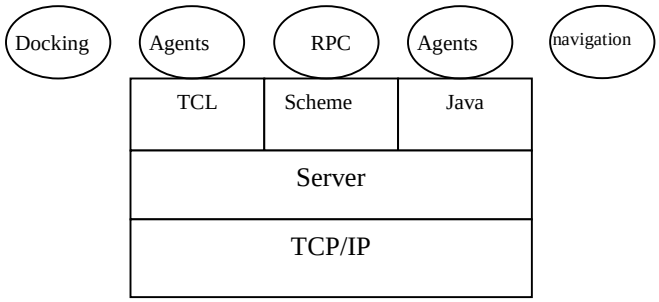

Figure 13: Agent TCL architecture [23]

The server level implements base level functionality. It is responsible [23] for keeping track of agents running on it, it providing low-level communication for the agents in the forms of message passing and streams, receiving and authenticating agents arriving from foreign hosts, and restarting an authenticated agent in an appropriate execution environment

The third level of the architecture consists of the executing environments that are currently supported. The Java section is simply a Java virtual machine with extensions. Any language can be added to Agent TCL by extending the interpreter to provide routines that capture the state of an executing program and provide an interface to the server level. The interface to the servers is in the form of an *agent_begin* and *agent_jump* semantics. The agent_jump will call on the state capture routines that, seize the local variables, and method and control stacks. This strong migration is required since portable computer devices are usually disconnected from a network.

While Agent TCL does provide strong mobility systems it must be emphasised that TCL is a scripting language which has limited use when describing complex data structures. An example of Agent TCL form is demonstrated in Figure 14 below. This agent hops between machine, executes the `who` command, and continues through its list of machines until it has visited them all, whereupon it returns home and displays the result. Although a very elegant script, which is more efficient that writing complex object oriented programs, it has no references or bindings to other agents or components. Capturing its program counter and execution state is a relatively trivial matter

```
agent_begin
set output{};
set machineList {host1, host2..}    //create an itenary
foreach machine $machineList {
      agent_jump $machine           // move to the destination
      append output [exec who]      // execute the 'who' command
}                                   // and append to the list
agent_jump $agent(home)
# display results
agent_end
```

Figure 14: TCL agent code sample

The final level of the architecture represents the agents themselves. They execute in the appropriate interpreter, using server-provided facilities for migration or low-level inter-agent communication. All other services are provided by dedicated service agents at the same level.

The dedicated service agents provide the main features that Agent TCL prides itself on. As mentioned previously, Agent TCL was designed for use on mobile hardware devices, hence an appropriate system needed to be devised for such an inherently disconnected entity. When an agent wishes to migrate to another node and the node is unavailable because, say, the present device is isolated, it sends the agent to a fixed machine which is permanently fixed. As soon as a connection to the desired node is available the agent is transfered. This *docking* machine (see Figure 15) is then charged with forwarding the agent to its destination.

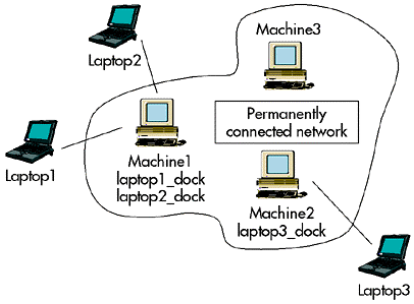

Figure 15: Laptop-Fixed Pair [23]

Each machine has a *dock master* agent that maintains the queue of migrating agents. These queues are always stored on disk to ensure that a machine crash will not lose the waiting agents. If the agent has an itinerary which includes other partially connected devices, its docking agent will not send it to a location X's docking agent in case that machine never docked. Such a scenario could leave an agent in exile waiting for that particular device to connect Instead the agent is only sent when the machine is docked.

The security agent's purpose is to protect agent servers from malicious agents. At the time of writing there was no mechanism in place which protected the agent from malevolent servers. Any agent or message sent between hosts is encrypted and signed to ensure security and authenticity. All resources (CPU, memory, disk etc.) have manager agents controlling access to them. A visiting agent is automatically untrusted and so run in an untrusted version of the appropriate interpreter. Requests for resource access are caught by a trusted interpreter which asks the resource's manager agent if the visiting agent can use it. The trusted interpreter then enforces any constraints that the resource manager agent imposes. For example the disk manager agent may only allocated 200K to the visiting agent thus preventing it from overflowing the disk space.

One other major service agent is the InterAgent Communication agent. It implements the high-level Agent PRC (APRC) system. ARPC can be used if the agent is local or at a remote location but since agents typically migrate to communicate locally the local case is used more. The system, like Voyager on static mode, requires interface definitions that are precompiled into stubs and proxies. The server stub registers with a nameserver agent that the client stub queries when looking for a particular agent. The nameservice agent is responsible for matching a client's interface request description with the server's provided interface definition. It will return a suitable agent even if it has more functionality than was originally requested.

3.5.3 Summary

Agent TCL was designed primarily with for use disconnected mobile computers. Its mobile agents (as the authors refer to them as) can be written in Tcl, Java, and Scheme although the public version only supports TCL. The TCL interpreter that hosts the agents loads the state of a migrating agent and restarts it, implying strong mobility. Communication [23] between agents is either low-level streams and message passing, or a high level RPC style mechanism. The docking system lets an agent transparently

jump off a partially connected computer (such as a mobile laptop) and return later, even if the computer is connected only briefly.

3.6 Jumping Beans

3.6.1 Overview

Ad Astra's Jumping Beans is a commercial Java framework that allows a developer to build mobile applications. The mobile application software dynamically jumps from host to host along with all of its essential information, including the code, data, state, resources, images, JARs, etc. so it can move to and execute on hosts that did not have the application previously installed. The unusual thing about Jumping beans, for mobile object systems, is the client/server architecture. Therefore any migration must go via the central server. The developers argue that this approach simplifies administration, enhances security and improves scalability.

3.6.2 System

As with Voyager and other commercial system it is not possible to get information on the inner details of the system. Instead an explanation of the system and its services is provided. The Jumping Bean system is a client server based system where all the mobile application passes through the server on every hop. All nodes [24] wishing to host a jumping bean need to host an *agency* client that that facilitates the receipt and dispatch of mobile applications, and enforces security.

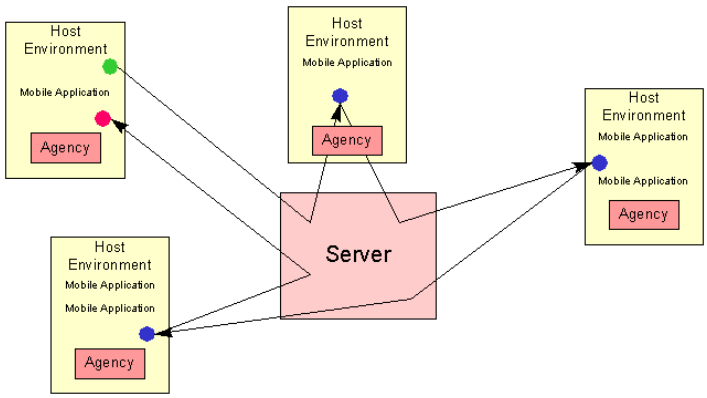

Figure 16: Jumping Bean System [24]

The agency facilitates the mobile application's discovery of the resident software in the host environment. All of the agencies supported by a server collectively form the "domain" of that server. Each agency belongs to exactly one domain, but one host computer can have multiple agencies from different Jumping Beans domains.

Mobilising [24] applications does not directly require a Client/Server design, but Jumping Beans relies on this approach in order to provide manageability and security. When a mobile application moves from one host to another, it first passes through the server, and then on to the destination agency. This allows the server to track all of the mobile applications within its domain. Since each agency will only accept mobile applications from the server in its domain, the server helps to enforce security.

Jumping Beans point to several key features as its strengths. In particular; manageability, easy integration into existing applications, transparent mobility, robustness and security.

The developer, Ad Astra, points to Jumping Beans' scalability despite it being a client/server system. Jumping Beans uses messages as a means of application communication. It argues that sending messages through a central server which knows the location of all mobile application is more efficient than the peer to peer approach. In a peer to peer system all mobile applications that migrate would have to leave 'scent marks' (essentially forwarding addresses) on any host they have visited. Thus, it is

argued, a message to another mobile application may end up being forwarded several times until it reaches its intended destination. This would usurp network resources and processing time thus proving to be an impediment to scalability. Garbage collection for scent marks would also be made more labour intensive.

Jumping Beans [24] has management capabilities built into it, such as centralised configuration management and tracking of mobile applications on the network. From the Advanced Server, an administrator can easily view, manage, and control the entire system from a central management console. The Advanced Server allows the administrator to track individual mobile applications in a real-time display, configure and revoke agencies, control mobile applications, and manage security configurations, all from the centralised server. The Jumping Beans Basic Server has a subset of these capabilities.

The system [24] was designed to be easily integrated into a development environment allowing mobile applications to be developed and dispatched without redesigning existing software. A Jumping Beans mobile application does not inherit from any parent classes, but instead implements a single Java interface: MobileApp. Since the system is Java based we presume it is based upon weak mobility but this is never specified. The mobile applications interact with the software resident on the hosts. Jumping Beans does not require that the developer work in any proprietary host environment, and does not require any proprietary distributed object environment, instead allowing the developer to use whatever is already familiar. Jumping Beans supports ordinary CORBA interfaces.

While normal CORBA dictates that the server remain in a fixed location (and therefore have a static IOR) Ad Astra have implemented a type of mobile CORBA which allows the server object to migrate yet still have clients making standard CORBA requests to these objects. Because of the central domain server this task is simplified as it monitors the location of all objects in its domain. It should be noted that mobile CORBA is not an actual specification of the Object Management Group.

With a central server, security is efficient and easier to enforce. Only those people authorised to use Jumping Beans can access it to build and/or dispatch mobile applications. Jumping Beans mobile applications use digital signatures to verify that they were dispatched from an appropriate server. Mobile applications are also authenticated using certificates and public/private keys. Each mobile application creates an audit log, which includes a log of the dispatching user and the hosts visited. All mobile applications can be tracked making it impossible to launch a “stealth” mobile application onto the network.

3.6.3 Summary

Jumping Beans is a commercial API that is manufactured by Ad Astra. Jumping Beans appears to have an inherent difference to systems reviewed to this point. Jumping Beans architecture is scalable because all messages get delivered directly by the domain server instead of following a trail left by a mobile application. A similar ‘follow-me’ mechanism is employed by the FarGo system (section 3.2) yet once the new location is discovered the message source updates its location tables. Essentially it will be a function of the amount of migration occurring as to whether a server provides a significant improvement. Mobile objects which are relatively ‘fixed’ to a site would nearly always be located at their presumed site.

3.7 Nomads

3.7.1 Overview

The University of West Florida’s NOMADS is a mobile agent system that aims to support strong mobility and provide an environment for safe Java agent execution. The creators have made strong mobility possible by creating an extended Java virtual machine that hosts the agent’s computational environment.

3.7.2 System

As already mentioned, the NOMADS system consists of a modified virtual machine, called the Aroma Virtual Machine, and an agent execution environment, named OASIS. Aroma [26] was designed with

the specific purpose of allowing multiple concurrent threads to be checkpointed (execution state of a process saved to some form of storage). The current checkpoint mechanism only supports saving and restoring the complete state of the virtual machine (including all running threads). Restoring the state is also possible only into a "blank" virtual machine (i.e., a virtual machine that has been initialised but does not have or had any threads of execution). Future versions will explore the possibility of restoring a checkpointed state into a virtual machine that is already running some threads.

Every agent executes within a separate agent execution environment, the Aroma VM. Each is separately instantiated, which gives control over resource accounting, and administrating but also causes increased overhead. Such a setup does not allow for the sharing of class definitions.

The Aroma execution environment executes in an OASIS process. Many OASIS processes can run on a host and multiple Aroma VM's can be executing in a single OASIS process. A port mapper module maps all OASIS processes on the host to their port numbers, allowing them to be individually accesses and identified.

An agent wishing to migrate to a remote OASIS process contacts the host's port mapper which returns the process' port number of the protocol redirector within the OASIS process. A protocol redirector exists in all OASIS processes. This module accepts connections from remote OASIS processes for agent transfer. A different protocol handler allows for communication with remote agents. A transfer protocol is agreed upon, although at present only a simple password authentication and state transmission protocol exists. The protocol redirector instantiates the appropriate handler which takes charge of negotiating with the remote platform. It receives the remote agent's state and then invokes a dispatcher module which creates and instantiates the agent. The dispatcher also enforces some polices set by the administrator (see below).

Since the agent has been checkpointed the migration is transparent. Conceivably this can occurs between two instructions (see Figure 17). The developers point to the ease of use afforded by this mechanism.

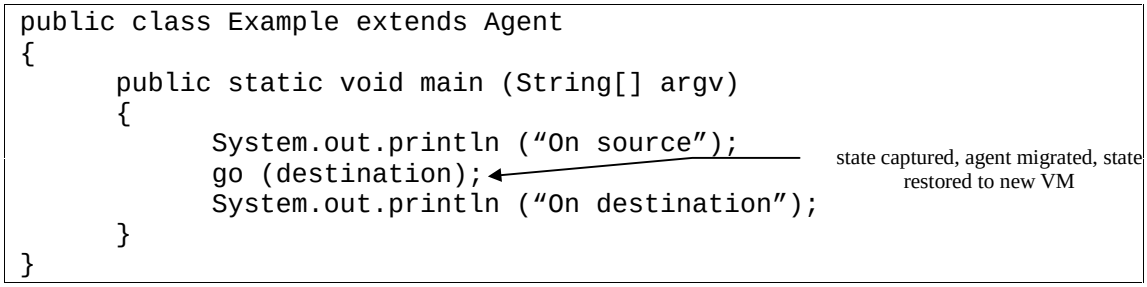

Figure 17: NOMADS based Mobile Agent [25]

The primitives [25] supplied can be split into two categories: mobility and communication. Mobility primitives are *go* and *send*. The difference being that *send* clones itself and sends the copy. Communication facilities are provided to allow access to the directory service which stores and agent id for a specified agent name. Finally *sendMsg* and *receiveMsg* allows inter-agent communication.

Along with the strong migration, NOMADS also provides a mechanism to establish security policies for all agents. People running OASIS processes have userid's. Policies can be setup around these individuals or groups can be established. Policies can then be established and enforced by the OASIS process. Policy categories include: authentication and agent transfer, execution (e.g. thread priority, execution duration), communication (e.g. max outgoing message size), access (using standard Java security managers), and resources usage (e.g. memory size, max thread count, network data transfer rate). These allow the administrator to prevent rogue agents from, say , consuming all the CPU time.

Another goal of the developers is to allow OASIS to dynamically adjust the resource limits of agents. An example might be the dynamic reduction (or increase) of the disk space assigned to an agent.

3.7.3 Summary

Nomads is a strong mobility mechanism for Java. Its environment is composed of and agent execution environment which runs in a modified Java Virtual machine. At the time of writing there was no actual version available to evaluate but the design that accompanied the original specification appeared to point to the possibility of full execution state transfer and safe agent execution.

3.8 Hive

3.8.1 Overview

Hive [38] is a research based distributed agents platform that was developed at he media lab at MIT. It was conceived as a decentralised system for building applications by networking local system resources. It is implemented completely in Java and relies heavily on Java's Remote Method Invocation API [34].

3.8.2 System

The Hive system is a collection of cells that host shadows and agents. A shadow is an interface to a local resource. The example given by the creators was that of a digital camera attached to a cell. The camera's shadow provides the access to the camera for any agents which may require its services, for example a `takePicture()` method. The shadows are static entities in a given cell. The do not migrate and do not communicate directly with the hive. However, the agents that reside on a cell are free to migrate between cells and use shadows locally or communicate with an agent on the cell which has access to the device's shadow. Cells are created as necessary to represent devices and are intended to be long-lived processes. They can run "server list agents" which contact registries in order to maintain relationships with other hives but this is not strictly necessary. By anology [38] to a conventional operating system, a cell is like a kernel, shadows are like device drivers, and agents are like processes.

An agent consists of the following: a Java object, an execution thread, a remote interface for network communication, and a self description which other agents can query in order to discover the agent's use and possible benefit. Agents can also export selected functionality over the network so that other agents can use its functionality.

Although the developers originally created Hive using Voyager (section 3.4) they soon discovered it limiting in flexibility and message passing. The system was then implemented using Java's remote method invocation (RMI) API that was more suitable for their needs. Voyager's attempts to provide a more transparent model of networking, such as the ability not to explicitly catch messaging failures proved to be ignoring something that was important to the systems robustness. I.e. the ability to address inevitable network failures.

Using Java RMI gives the system a weak mobility paradigm which the authors acknowledge would be better suited to strong mobility methodologies.

The system also uses code mobility in order to dynamically update various components but with this benefit came the drawback of dealing with versioning. This is never properly addressed in Hive.

An ad-hoc approach is taken to agent communication with individual agents deciding how to talk to each other. This, as already mentioned previously, is achieved using RMI, which is synchronous. To allow agents to decouple themselves from other agents, especially for event notification, asynchronous calls to RMI where added. Agents have two types: the true type and the remote interface. The true type is a subclass of `AgentImpl` while the remote interface subtypes `Agent`. This distinction lets agents have administration methods that are protected from calling by other agents.

Since Hive agents can offer services to other agents it is necessary that they have some form of describing themselves so that ad-hoc agent interaction can occur. For this the designers have provided two separate ontologies so agents can advertise: Syntactic and semantic.

Syntactic ontologies are provided by the strong type system in Java. Given a reference to an agent, it is easy to learn the types it supports and, therefore, the messages it understands.

Semantic ontology compliments the limiting syntactic ontology of Java. Hive utilises the Resource Description Framework, which is built on XML to provide a structured way to attach nouns and verbs for agents. The agent's semantic description could include its physical location and an obvious nickname.

The infrastructure put in place by Hive is specifically designed for 'things that think' domain. This vision is one where computation devices are embedded in the most mundane everyday objects such as table tops, household appliances and clothes. Kitchen helper agents could dispatch one agent to the fridge hive which could return the contents, while this information could be sent to a recipe book hive which in turn dispatches an agent to the preparation counter in the kitchen. Now the resident has access to a recipe, the ingredients of which are sure to be in the fridge.

Although the example given above may seem a little flippant, it is a genuine application of the potentials of the inevitable ubiquitous computing. Hive is the basis of a very practical use of mobile agent/object systems.

3.8.3 Summary

Hive is a Java realised mobile agent system that implements weak mobility. It is a system for building applications through the networking of local resources. All remote communication is through standard Java RMI which the developers have taken to be sufficiently robust to cope with the possibility of large-scale deployment.

3.9 Other Systems

3.9.1 Sumatra

Sumatra [33] was a part of a larger project developed at the University of Maryland (as distinct from the discontinued Sumatra project at University of Arizona which examined an infrastructure for experimenting with mobile code). The project focused on how mobility can be used as a tool that programs use to adapt to variations in network characteristics. The work used an Internet chat program as the basis for its experiments, primarily because it is highly interactive and requires fine grain communication. Being aware of network latency, the chat server can move to a different node. Such an application can, with the appropriate resources, migrate to a position that better leverages the supporting infrastructure, in turn providing a better service to the end user.

Migration decisions are made by application specific policies and Komodo, a distributed network latency monitor. These are beyond the scope of this paper.

The mobility part is named Sumatra, which is an extension of the Java programming environment. Two programming abstractions are introduced, object-groups and execution engines. Object groups (OG) appear to be similar to FarGo's complets; objects are added or removed from object groups, which are then treated as the unit of mobility. The execution engines are also a similar concept to FarGo's Core; OGs move between execution engines. Upon an OG's migration all local references it has to objects in different OGs are converted to proxy references. Similarly, all references to the migrating OG are also changed to proxy references. The system uses the type information stored in the class template to achieve RPC functionality without needing a stub compiler.

Thread migration is explicitly invoked using an `engine.go()` function. All thread stacks in Sumatra have a parallel type stack that is used to identify references to other objects. All local objects which are not members of OGs are transported with the migrating thread. Any objects that belong to an OG have their references in the migrating thread's value stack converted to proxy references.

Sumatra migrates objects in a manner that may cause problems for applications other than an Internet chat server. Such an application requires bindings to end users to be maintained. However, in other application domains such as e-commerce or information retrieval, retaining proxy references to objects

that may be spread arbitrarily around the world may not be desirable. The inherent latency issues over large distances and inability to detect faults reliably point to a technique that strives to only retain such proxy references to that which is either essential or desirable.

3.9.2 Telescript

Developed by General Magic, Telescript is an object-oriented language conceived for the development of large distributed applications. It is an open, object-oriented, remote programming language [26]. There are three simple concepts to the language: agents, places and "go". Agents "go" to places, where they interact with the place or other agents to get work done. Telescript contains mechanisms to ensure the safety and security of mobile programs and the servers that host them.

Telescript [5] employs an intermediate, portable language called *Low Telescript*, which is the representation actually transmitted among engines, the Telescript computational environment. Engines are in charge of executing the agents. Telescript servers [4], which are called *places*, offer services, usually by installing stationary agents (akin to Agent TCL) to interact with visiting agents. Places are essentially stationary agents that can contain other agents.

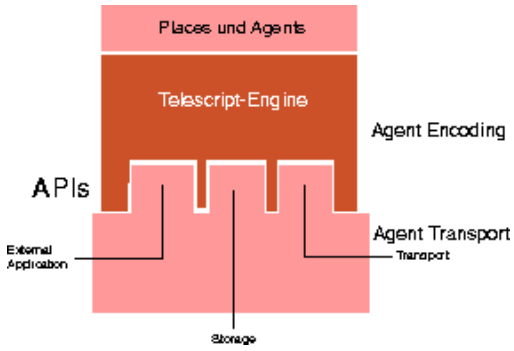

Figure 18: Telescript Structure [4]

Telescript implements strong mobility so execution resumes immediately after the *go* primitive has been issued. Relative migration is also possible using the *meet* primitive. Collocated agents can invoke each other's methods for communications and an event signalling facility is also available.

Each agent and place has an associated *authority*, which is responsible for them. A place can query an incoming agent's authority and potentially deny entry to the agent or restrict its access rights. the agent receives a *permit*, which encodes its access rights and resource-consumption quotas. The system terminates agents that exceed their quota and raises exceptions when they attempt unauthorised operations.

Telescript agents needed to be part written in C and part written in Telescript. This was the primary reason that it was commercially unsuccessful. General Magic shelved the project in favour for a Java-based system named Odyssey that uses the same design framework. It does not feature strong mobility.

3.9.3 Concordia

A Concordia system [27], at its simplest, is made up of a Java VM, a Concordia Server, and at least one Agent. Concordia servers are Java application which play host to agents. The server manages code, data and movement.

Usually, there are many Concordia Servers, one on each of the various nodes of a network. The Concordia Servers are aware of one another and connect on demand to transfer agents in a secure and reliable fashion. The agent initiates the transfer by invoking the Concordia Server's methods.

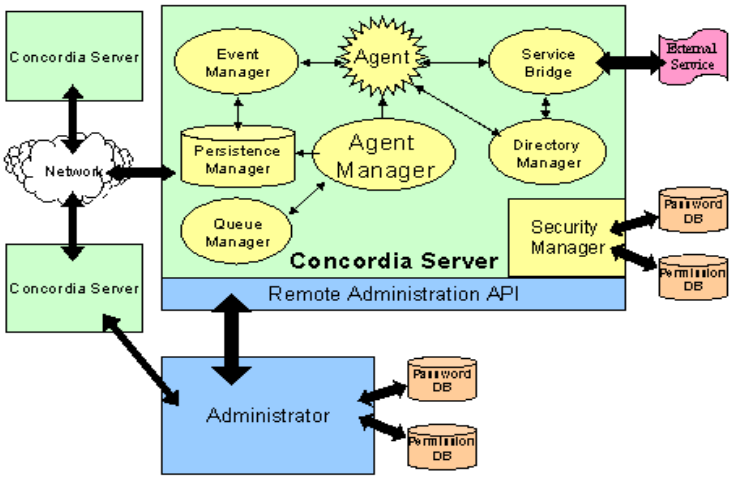

Figure 18: Concordia Architecture [27]

This signals the Concordia Server to suspend the agent and to create a persistent image of it to be transferred. The Concordia Server inspects an object called the Itinerary, created and owned by each agent, to determine the appropriate destination. That destination is contacted and the agent's image is transferred, where it is again stored persistently before being acknowledged. In this way the agent is given a reliable guarantee of transfer.

After being transferred, the agent is queued for execution on the receiving node. This happens promptly but possibly subject to certain administrative constraints. When the agent resumes executing, it is restarted on the new node according to the method specified in its itinerary, and it carries with it those objects that the programmer requested. Its security credentials are transferred with it automatically and its access to services is under local administrative control at all times.

Condordia Agents, like most programs, generally have several components. An agent might start interactively, by prompting the user for search information, then may travel to a server to perform the query. Or, the agent may simply be a kind of remote demon, such as a mailbox filter or notification sender. As its methods complete, the itinerary causes the agent to be moved to other Concordia nodes. Therefore agents with different purposes will typically have different itineraries.

In all cases, the Concordia agent is autonomous and self-determining in its operation. In this way, it is unique since it is in control of its own itinerary.

The Concordia system is made up of numerous components, each of which integrates together to create the full Mobile Agent framework. The Concordia Server is the major building block, inside which the various Concordia Managers reside. Certain Concordia components have a user interface, such as the Concordia Administrator. In any case, each Concordia component is responsible for a portion of the overall Concordia design, in a modular and extensible fashion.

4 Summary

After the mainframe era came the networked client server paradigm. This extra connectivity spurred the remote procedural call, allowing the execution to flow across machines with execution eventually returning to the original location. Migration of the executing thread required new concepts. Since no one particular programming language was omnipotent in servicing mobility, designers often worked from the ground up, designing new paradigms and systems. They often proved unpopular because of the need to adapt to its syntax rather than integrating them into existing languages. The consequence was typically non acceptance. They did however give valuable insight into the generic problems inherent in mobility.

The debate between strong and weak mobility is still argued over. Popular reading lists demonstrate the lack of consensus on this issue. People are divided over whether there is, or will ever be, a killer app that uses strong mobility. Weak mobility, it is said, provides an more suitable method of migration. Migrating sections of a thread's stack (as in the case of Emerald) gives reason for concern. Any node, which hosts part of the stack, is liable to fail at any time thus rendering the remainder of the thread in limbo.

In the context of virtual enterprises and intelligent buildings we require a system which enables mobile objects to move freely around a system yet have bindings to services dynamically maintained. Foundational to the operation of mobility in distributed applications is object binding. Fuggetta describes binding as the act of attaching to a resource by means of an appropriate type of reference. Of the systems studied here FarGo provides the most concrete example of Fuggetta's abstraction. Binding to other mobile objects can be made using special references which dictate co-location and re-location rules.

The systems reviewed have similarities in the features they offer. Table 1 gives a summary of main features. A common trend is for modern mobility frameworks to be written in Java. Java's network rich features and serialisation techniques along with its popularity make it the language of choice in the development community.

System	Commercial/ Research	Mobility Type	Underlying Technology	Security	Event Model
Emerald	R	strong	Proprietary (Emerald)	none	N
FarGo	R	weak	Java	limited	Y
Aglets	C/R	weak	Java	Mutual authentication. Aglets protected by proxy	Y
Voyager	C/R	weak	Java	Server protected using *trusted* and *untrusted modes*	N
Agent TCL	C	strong	TCL / Scheme / Java	PGP communication, Safe TCL used as secure execution environment	N
Jumping Beans	C	weak	Java	Central server manages all security	N
Nomads	R	strong	Java	Policy Based security protocol mechanism	N
Hive	R	weak	Java	N/A	JINI
Sumatra	R	strong	Java	N/A	N
Telescript	C	strong	Proprietary (Telescript)	RSA Secure intra agent comms, server imposes quotas on visiting agents	N
Concordia	C	weak	Java	SSL intra agent comms, ACL based server protection	Y

Table 1: Main features of reviewed systems

While Java threads are objects, Java has no mechanism to capture state at the thread level. Serialising an object will only capture the non-transient and non-static global instance variables. Attempts to rectify this require either a modified virtual machine [25], which allows thread state to be captured and saved, or by the use of a pre-compiler processor [28] which inserts appropriate state acquisition semantics. These techniques generally result in an incompatible virtual machine or a class with large amounts of state-capturing code.

4.1.1 Extensibility

In terms of extensibility, older proprietary systems are inclined to be incompatible with existing ones. Emerald was never intended to be running enterprise applications on global WANs. Rather, it demonstrated the viability of transferring executing processes to different address spaces on different nodes using the Smalltalk [29] object oriented programming model. Another important discovery of this research was that moving objects which are used by processes are not simple problems. There is no 'best' solution for distributed computing. Like all difficult problems there are lots of compromises.

Telescript was dropped by General Magic in favour of new Java version (without strong mobility) called Odyssey. New commercial and development systems are practically all being implemented in Java because of its extensibility. Java also has the advantage of being designed specifically with the Internet in mind. It has the unparalleled advantage of coming into existence at the same time as the Internet began to move from academic and military circles to the civilian world. Voyager has received multimillion-dollar Department of Energy grants to further pursue Voyager development. This further encourages Java R&D.

When Java was originally launched columnists at the time were uncertain as to its future but Sun's licensing agreements allowed people access to a sophisticated and convenient object oriented language for free. Java 2 allows for further extensibility with the facility to extend the virtual machine with, for example, enterprise beans. As a rule, Java based system are fully extensible. Virtual Machine source code is available should any particular part need re-engineering but this may be at the price of portability.

4.2 Conclusion

The reviewed systems all provide a particular method implement mobility. Different systems are designed to address particular problem domains. No one system provide a perfect solution to every problem. Using either the strong or weak migration paradigm, objects move between hosts and get reactivated. However despite this there is a case to be argued that a specific area has not been properly addressed. In terms of long lived, large scalable systems over the Internet, the reviewed systems which retain bindings to previous hosts, will not provide a strong migration system reliably.

Object oriented strong mobility over the Internet, in its current guise, is destined to fail. Most strong mobility systems are intended to run over clusters or high speed LANs where the underlying medium is fast and reliable. Taking Emerald as an example; when an object is moved to another node the activation records, or stack segments, of threads which are running the object's methods are moved with it. For a thread to return to the migrated object's method it must pass control to the foreign node. Essentially, the return address for the method on the next frame down must be redirected to another node. Mapping this paradigm onto a WAN or the Internet and expecting the same outcome is to ignore the new observables that WANs introduce.

Wide area networks stretch across the globe and in the medium term future they may even stretch further. The speed of connections over large physical distances are limited by the finite speed of light which, in terms of real time control can still have unacceptable lag. A good example of the actual delays can be seen when television programs have satellite link-ups to between continents. The conversation is interlaced with short pauses that prevent normal conversation. A large part of this delay is introduced by the various electronic components between the links and not by the microwave linkup between satellite and ground station. Real time control systems [37] can not work with

(relatively) large delays, they are destined to become unstable. The same is true for a strong mobility system operating over long distance. Although delay may only be milliseconds, in terms of computing speed where main memory operates at the nanosecond level and processor registers are taken to be instant, these lead to an unstable system whose problems are exasperated by bandwidth fluctuations and unpredictable failures.

Although it is possible to introduce a quality of service (QoS) [35] element into the networking infrastructure. In a democratic network, where all packets are equal, congestion will occur in overload situations leading to fluctuations in bandwidth. These swings are unpredictable so prove more troublesome to strong mobility applications; if the application was aware of delays over certain distances it could take these into account.

Furthermore, WANs have trouble revealing failures [35]. When a call to a function on a remote node does not return, several presumptions can be made. It may be a busy node where the request has been queued, the node may have crashed, the process may be deadlocked, or the network infrastructure may have failed at some point. Whatever the cause, it is impossible for the originating node to make an informed decision on the most appropriate action to take.

The three postulates of WANs, physical location, bandwidth fluctuations and failure discovery, alludes to the inherent flaw of merely changing references from local to remote without an appreciation of the underlying unreliable infrastructure.

Take the abstraction of an *object–group* (Sumatra) or *Complet* (FarGo) where several objects are grouped together in mutually beneficial clusters. In Java, threads and objects are separate entities. A Java thread is created from object's methods but then becomes an abstract entity onto itself. The state of the object is retained separately. Executing threads hold all local variables generated while executing and invoking methods. The section of memory they occupy is referred to as the working memory. All objects have state encapsulated in the shared main memory, which is split between static and dynamic memory.

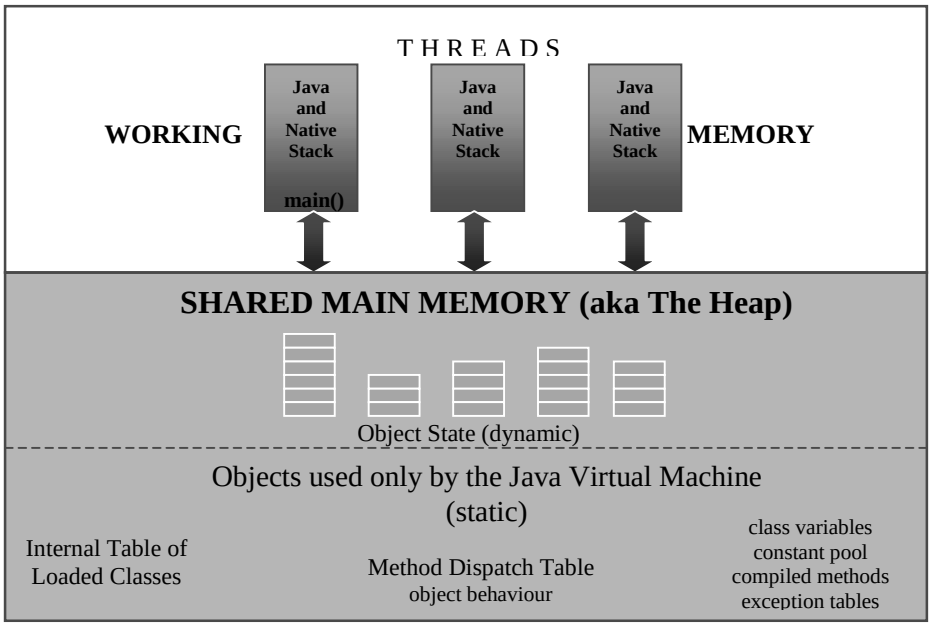

Figure 19: Java Virtual Machine Internals (HotSpot technology) [36]

Strong mobility (for Java in this case) involves migrating the execution state of a thread (its stack), its object state and the classes bytecode. Sumatra also retains a type stack in parallel to all thread stacks. Since data on the stack has no type information encapsulated with it, this allows references to other objects to be quickly identified. Any such bindings to objects belonging to other object groups are

transformed to proxy references. The design of the system is such that retaining bindings is desirable but for large systems, which are widely distributed, these bindings can become an encumbrance.

4.2.1 Future Work

Design of strong mobility systems, which will work over the Internet, is an area that has not received a great deal of attention from the research community. Current methods of implementing strong mobility are not suited to the Internet. The retention of bindings to other objects is a fundamentally flawed idea when implemented over WANs.

Instead we propose a system which uses a combination of techniques to achieve 'as strong as possible' mobility. The three techniques to accomplish this are:

- Stack Collapse
- Data Space Management
- Isolated Thread migration

An application designer writing applications that allow the `main()`(or similar) method to be returned to removes all local variables from thread stack which usually prove so awkward to capture. The object's state is then fully encapsulated in its instance variables, which can be easily serialised and migrated and restarted. The area will require more detailed attention in order to turn a theory into a practice. However certain objects may not be serialisable. If this is the case an effort must be made to find an alternative on the destination site instead of resorting to retaining a network proxy.

An abstraction of a container is envisioned into which all threads and objects, which are mutually beneficial, are placed. While certain threads and objects within the container will reference objects outside the container in some cases it may be possible to find an alternative on the destination node which is being migrated to. For example, if a node has an ORB, which is available for use by all objects, it is possible to break the reference to it and rebind to another ORB on a destination node. Using this method we have removed the need to retain an inefficient network reference. This is referred to as data space management by Fuggetta [5].

The final stage of migrating is the case where a thread in the container is doing isolated work and is not referencing or being referenced from outside its container. A simple example would be that of a thread which is performing a matrix multiplication or any similar introspective work. Requesting this thread to stop and restart executing would only result in time wasted and information lost. A thread such as this is a prime candidate for strong migration as it has no 'loose ends' that need to be converted to network references. Using Java's Just in Time (JIT) API it is possible to grab the state of such objects. Future work will be concentrated on reseeding these threads a different virtual machine.

The combination of the three techniques should provide a framework that will attempt to give strong mobility to threads which are suitable, while servicing all other threads in an attempt to reduce the network references to zero. Essentially network references should only be retained if they are essential or desirable.

References

[1] Luca Cardelli, “Abstractions for Mobile Computation”, Secure Internet Programming: Security Issues for Mobile and Distributed Objects. Lecture Notes in Computer Science, Vol. 1603, Springer, 1999.

[2] Luca Cardelli, “Mobile Computation”, Mobile Object Systems - Towards the Programmable Internet. Lecture Notes in Computer Science, Vol. 1222, Springer, 1997. pp 3-6

[3] E. Jul, H. Levy, N. Hutchinson, A. Black, “Fine-Grained Mobility in the Emerald System” ACM Trans. On Computer Systems, Vol. 6, no. 1, pp. 109-133, February 1988.

[4] Neeran M.Karnik, Anand R. Tripathi, “Design Issues in Mobile-Agent Programming System”, IEEE Concurrency, pp 52-61, July - September 1998

[5] A. Fuggetta, G.P. Picco, G. Vigna, “Understanding Code Mobility” IEEE Trans of Software Eng., vol 24, no. 5, pp. 342-361, May 1998.

[6] Jeff Nelson, “Programming Mobile Objects with Java”, Wiley Computer Publishing

[7] V.J. Cahill, P.A. Nixon, F.A. Rabbi, “Object Models for Distributed or Persistent Programming”

[8] Ophir Holder, Israel Ben-Shaul, “Dynamic Layout of Distributed Applications”, Dept. of Electrical Engineering, Israel Institute of Technology.

[9] Ophir Holder, Israel Ben-Shaul, Hovav Gazit “Dynamic Layout of Distributed Applications in FarGo”, Proceedings of the 21st International Conference on Software Engineering, May 1999.

[10] Ophir Holder, Israel Ben-Shaul, Hovav Gazit “System Support for Dynamic Layout of Distributed Applications ”, Dept. of Electrical Engineering, Isreal Institute of Technology

[11] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall, “A Note on Distributed Computing”, Technical Report TR-94-29, Sun Microsystems Laboratories, Inc, November 1994.

[12] Object Management Group, “The Common Object Request Broker: Architecture and Specification. Revision 2.2”, February 1998

[13] Sun Microsystems, Inc, “Java Remote Method Invocation (RMI) Specification”, December 1997

[14] Richard Grimes, “Professional DCOM Programming”, WROX Press, 1997

[15] D. E. Lenoski, J. Ludon, K. Gharachorloo W-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, M. S. Lam, “The Stanford DASH Multiprocessor”, IEEEC, Vol 25, No. 3, pp 63-79, March 1992.

[16] ObjectSpace, Inc., “ObjectSpace Voyager 2.0 Core Technology User Guide”, 1997 *available from* http://www.objectspace.com/

[17] Simon Dobson, “Mirgants – a brain-numbingly simple mobile code system” *available from* http://www.cs.tcd.ie/Simon.Dobson/migrants/

[18] D.B.Lange, “Java Aglets Application Programming Interdace (J-AAPI)”, IBM Corp., white paper, Feburary 1997

[19] Mitsuru Oshima, Guenter Karjoth, Kouichi Ono, “Aglets Specification 1.1 Draft”, IBM Corp., Specification, September 1998

[20] Joseph Kiniry, Daniel Zimmerman, “A hands-on look at Java mobile agents”, IEEE Internet Computing, Jul/Aug 1997.

[21] OrbixWeb Programmer’s Guide, IONA Technologies PLC, September 1998

[22] http://www.daimi.aau.dk/~bouvin/otw/index.html

[23] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, George Cybenko, “Agent TCL: Targeting the Needs of Mobile Computers”, IEEE Internet Computing, Vol 1, No. 4, July/Aug 1997

[24] Ad Astra Engineering Inc., “Jumping Beans White Paper”, May 1999. *available at* www.jumpingbeans.com

[25] Niranjan et al, “NOMADS: Toward an Environment for Strong and Safe Agent Mobility”

[26] http://www.w3.org/Search/9605-Indexing-Workshop/Papers/Friedman@GenMagic.html

[27] Mitsubishi Electric ITA Horizon Systems Laboratory, “Mobile Agent Computing”, White Paper. *available from* http://www.meitca.com/hsl/Projects/Concordia/documents.htm

[28] Stefan Fünfrocken, “Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs)”, Proceedings of the Second International Workshop on Mobile Agents (MA'98), Stuttgart, Germany, September 9 - 11, Springer-Verlag, LNCS 1477, pp. 26-37, 1998

[29] David A. Taylor, “Object-Oriented Technology: A Manager's Guide”, Addison-Wesley, Reading, MA, USA, (1993) *available from* http://www.whatis.com/smalltal.htm

[30] Michael Morrison, Randy Weems, Peter Coffee, Jack Leong, “How to Program JavaBeans”, Ziff-Davis Press, 1997

[31]] S.Green, L.Hurst, B.Nagle, P.Cunningham, F.Somers, R.Evans, “Software Agents: A Review”, Trinity College, Broadcom Eireann Reasearch Ltd. *available from* http://www.cs.tcd.ie/publications/tech-reports/tr-index.97.html

[32] Mitsuru Oshima, Geunter Karjoth, Kouichi Ono, “Aglets Specification 1.1 Draft”, IBM corp 1998.

[33] M.Ranganathan, Anurag Acharya, Shamik D. Sharma, Joel Saltz, “Network aware Mobile Programs”, Proceedings of the USENIX 1997 Annual Technical Conference, Jan 6-10, 1998

[34] Java Remote Invocation - Distributed Computing For Java, White paper, Sun Computer Corporation

[35] Luca Cardelli, “Wide Area Computing” Automata, Languagese and Programming, 26th International Colloquium, ICALP'99 Proceedings. Lecture Notes in Computer Science, Vol. 1644, Springer, 1999. pp. 10-24

[36] Douglas Kenneth Dunn, Java Rules! Electronic Book, 1999 *http://www.javarules.com*

[37] Franklin, Powell and Emani-Naeini, 'Feedback Control of Dynamic Systems' 3rd ed. Addison-Wesley 1994, pp568-573

[38] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, Pattie Maes “Hive: Distributed Agents for Networking Things”, Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents.