As strong as possible mobility: An Architecture for stateful object migration on the Internet

Tim Walsh, Paddy Nixon, Simon Dobson

Distributed System Group, Department of Computer Science
Trinity College Dublin
Email: {Tim.Walsh, Paddy.Nixon, Simon.Dobson}@cs.tcd.ie

ABSTRACT: A major challenge for distributed applications working in mobile contexts is to provide application developers with a way of building stable systems whose elements may change across time. We present a design strategy for managed mobile distributed applications in which we clarify the relationship between algorithm and location management. We introduce the concept of *As Strong As Possible* mobility that uses a combination of data space management and thread state capture so that objects and thread can migrate in a manner that has not been explored yet. The ultimate goal is to provide a mechanism for mobility where an object will be migrated using strong mobility techniques where possible and using rebinding mechanisms when it is not meaningful to simply 'grab' a thread's state.

Keywords

Migration, strong mobility, data space management, Java

1 Introduction

Internet-distributed systems are beginning to offer a serious platform for stable, long-lived, flexible applications development. There is an increasing realisation that such 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. The scale and dynamism of this type of Internet application is exemplified in the ideas of teleworking or virtual enterprises. In these situations individual users or corporations must interact with rapidly changing collections of other workers or corporations in a secure and flexible manner. Issues such as the current location of the worker, trustworthiness of the worker, and access to data must be managed in addition to the normal function of the application being utilised. A major challenge for such distributed applications is to provide stability in the face of constant change, providing application developers with a stable architecture whose elements and locations may change across time.

We are particularly concerned with building *virtual enterprises* [6] (Internet-scale mission-critical distributed systems spanning a changing number of component organisations and administrative domains) because such systems are the essential infrastructure for projects ranging from electronic commerce to large-scale scientific collaborations. We examine new mobile object concepts that can serve as an infrastructure for these domains.

Current solutions to decentralised applications alone will not service the needs of this type of highly dynamic and mobile application. What is required is an inherently mobile solution detached from the present client-server model and the associated problems of scalability and flexibility [1]. These solutions will exploit mobile code to enable bindings between code fragments and their resources to be changed dynamically [2].

Present code migration techniques offer two extreme solutions to code migration; weak and strong. We present a design strategy for managed mobile distributed applications in which we clarify the relationship between algorithm and location management. At present, systems programming languages that implement strong mobility are designed for tightly coupled distributed systems. This paper presents a pseudo-strong mobility mechanism for global or loosely–coupled networks.

2 Mobility issues

Existing enterprise distributed systems are typically client-server. The client-server paradigm permits program execution to pass temporarily to a remote address space. This allows servers to offer services to remote nodes.

A logical advance on client-server solutions was to free the executing process from its originating node thereby making the code mobile. Benefits of mobile code [3] include:

- Load sharing: Mobile objects can take advantage of underused processors.
- Communication performance: Objects that use the network intensively can reduce communication cost by migrating to the foreign host for the duration of the interaction.
- Reconfiguration: Migrating objects permit continued service during upgrade or forecasted node failure.
- Availability: Objects may be moved to improve the service and provide better failure coverage.

An executing process is essentially the encapsulation of code and execution state. The execution state represents the data upon which the code operates. Transferring both elements from one node to another, on face value, seems trivial. Executing processes can reference other entities using pointers or, as is typical in object oriented systems, object references. These references must be managed in a consistent and coherent manner. Managing these references is the main problem as will be demonstrated in later sections.

John Ousterhout's dichotomy divides high-level languages into system programming languages and scripting languages. System programming languages (or "applications languages") are strongly typed, allow arbitrarily complex data structures, and programs in them are compiled, and are meant to operate largely independently of other programs. Prototypical system programming languages are C and Modula-2.

By contrast, scripting languages (or "glue languages") are usually weakly typed or untyped, have little or no provision for complex data structures, and programs in them (called scripts) are interpreted. Scripts need to interact either with other programs (often as glue) or with a set of functions provided by the interpreter, as with the file system functions provided in a UNIX shell and with Tcl's GUI functions. Prototypical scripting languages are AppleScript, C Shell, MSDOS batch files, and Tcl.

2.1 Mobile Objects and Mobile Agents

The term agent is one of the more overused words in computer science and has different meaning in the area of artificial intelligence and distributed systems. According to Nelson [9] the key different between the two is autonomy. Mobile objects have no autonomy. They can be fetched from the destination or sent from the source but upon arrival they remain dormant.

Mobile agents have autonomy. The method used to reach the destination site may be the same as mobile objects but upon arrival they resume or restart execution. This requires a server or daemon at the destination that will act as a docking station or 'sandbox' for the agent. The sandbox should provide the 'needs' of the agent. The main difference between mobile agents and objects here is the part that gives execution back to the agent. An everyday example of this in action is a web browser that fetches a class files from a server and then runs. It runs in a browser's sandbox for security reasons. Access to key system components is forbidden.

Voyager [11] has a facility for mobile agents. A voyager daemon waits on a predetermined port for a mobile agent. When one arrives it unpacks it and calls a method that has been specified prior to migration. IBM's Aglets [11] provide a similar agent mechanism.

There is a slight inconsistency with the concept of mobile agents when examining Cardelli's definition of an agent. He states that agents [1] are assumed to be completely self-contained. They do not communicate remotely with other agents, rather they move to some other location to communicate locally. While this holds true for system like Voyager and Aglets it is conceivable to have the same systems operating a type of data space management system as described by Fuggetta in [2].

Objects based application use other objects, sometimes termed resources, all the time. They are linked or bound together by object references. Hive [19] demonstrates this with the notion of agents and shadows. Shadows are interfaces to actual physical hardware such are printers and digital cameras. The shadow for the camera therefore would have a `takePicture`() method as part of the interface. An agent can move to another node yet still use the shadow of the camera from a distance. This is clearly something that contravenes Cardelli's definition. The agent is not travelling to a host to use a resource, instead it uses, in the case of Hive, Java's RMI to communicate. Upon migration of this agent it should be possible

to continue to utilise the camera's shadow. It requires the service to be rebound to upon the agent's arrival at its new host.

What this establishes is that the area of mobile objects and agents is in its infancy. It has had no rules or standards placed upon it, unlike the client server paradigm. No one system has been created which will satisfy all needs and it is very doubtful whether it is possible to create such a system as different application domain had varying requirements.

2.2 Extent of State Migration

In the above example of an applet being executed within a browser established that the granularity of state migrated can be very limited. The applet will contain, at most, initialisation data. This type of migration is classified as weak mobility. **Weak mobility** [2] 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.

At the other end of the spectrum lies strong mobility. **Strong mobility** [2] (or strong migration) allows migration of both the code and the execution state to a remote host. We are specifically examining system programming languages. While systems such as Agent TCL [18] do 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 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 would be far 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. It should be noted that such a program could be implemented, albeit more convolutedly using a weak mobility Java system such as IBM's Aglets [9]. There are no complex data structures and no references to other agents that could cause confusion. It is a self-contained entity.

```
agent_begin
set output{}; set machineList {host1, host2..}
foreach machine $machineList {
      agent_jump $machine
      append output [exec who]
}
agent_jump $agent(home)
# display results
agent_end
```

Figure 1: Agent TCL code sample

Strong mobility has been chiefly fixed in the domain of system programming languages and proprietary systems such as Emerald [3]. These systems were specifically designed to run on small LAN's and clusters. In the case of Emerald, if an object is migrated, all threads that have stack frames pertaining to that object must be moved with it. When the stack shrinks it will come to the part of frame that is now hosted on a foreign node. At this stage execution of this thread will continue on the foreign node before returning to its original home. See figure 2. Such a design is feasible when there is a fast, reliable network backbone but completely unsuitable when operating over the Internet. Even [7] with fibre optic links the network's speed is limited to the speed of light which, in computer terms, can be relatively slow over very large distances. Faults on the network are practically impossible to isolate. These shatter any attempt to give the illusion of transparency that is achievable on efficient cluster computers.

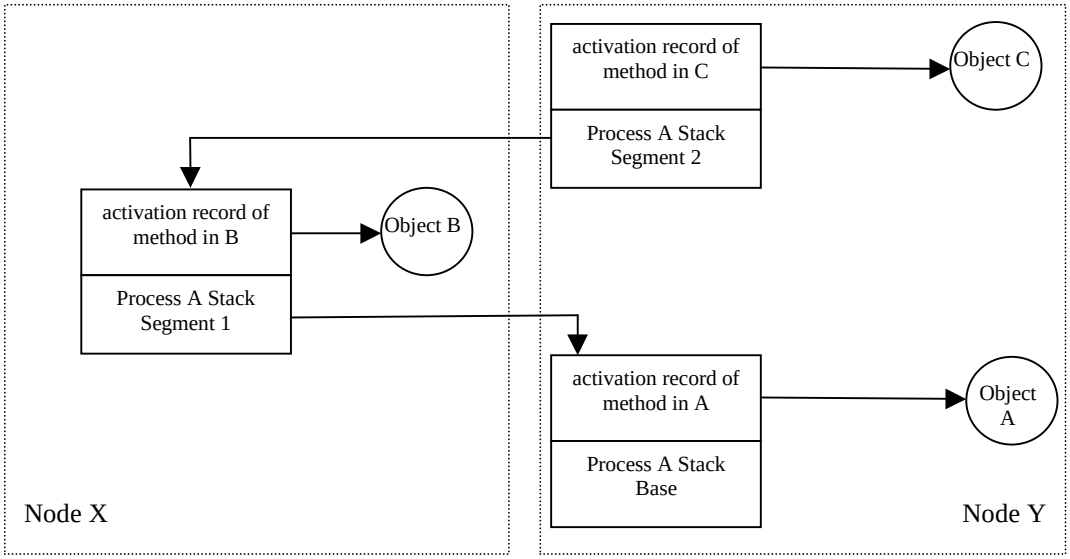

Figure 2: Emerld thread migration [3]

2.2.1 Advantages and Disadvantages

The advantage of strong mobility [4] is that long-running or long-lived threads can be suddenly move or be moved from one host to another. In principle, such a transparent mechanism would allow continued thread execution without any data loss in their ongoing execution. Such an approach is useful in building distributed systems with complex load balancing requirements. These threads need not explicitly know about their movement.

Unfortunately large distributed networks are bad sites to run systems which require inherently coherent coupling to other nodes. Waldo [8] argues this case in point. To attempt location transparency over large distant networks is attempting to ignore latency, memory access into different address spaces and partial failure. Such lessons have been learned primarily from NFS, which was the first main attempt at large distributed file systems. Implementing strong migration over a slow unreliable medium using existing techniques is guaranteed to develop serious problems. Clusters and high speed networks offer a better platform to run these systems.

Weak mobility does not involve any migration of execution state. If Emerald employed weak mobility it would not transfer any activation records when the object moved. Instead, although the object and its state would move, the stack frame for process A would stay intact.

A modern day instance is the Java serialisation mechanism. When an object is serialised, all the instance variables of the object are packaged into an output stream. This data can then be shipped with a copy of the class bytecodes to another node and reinitialised with the same state. No execution state has been transferred. Threads on the original source node continue to use the code.

The Observer [20] pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically. This was the major influence behind the Actor pattern. The Actor [9] design pattern can be used to migrate execution state in a somewhat unconventional manner. Figure 3 shows the dependencies. The Actor is an implementation of an object that encapsulates one or more threads. The environment is an abstraction for anything the Actor might interact with or know about. Using this pattern decouples the thread from the environment.

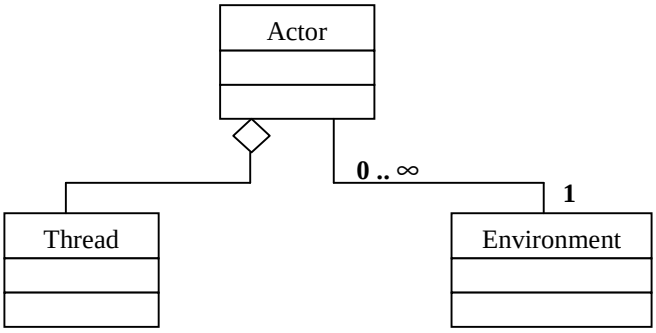

Figure 3: Actor Pattern [9]

The Actor controls access to the environment that prevents extrinsic state about Thread in Environment. The classes in Figure 4 and Figure 5 show how using this pattern can achieve a thread migration. When MobileObj is serialised, this tries to serialise the LoopThread object. As the class in Figure 5 is a specialised thread it is still not possible to encapsulate the thread's state using serialisation. Instead the externalisation interface is used which calls `writeExternal().` The writeExternal method suspends the thread and bundles all info relating to the thread. All the variables of the thread are specifically written to an output stream.

```
import java.io.*;

public class LoopThread
                extends Thread
                implements java.io.Externalizable {

        public String message = "and going";  // Count number of "going" messages

        public LoopThread(String m) {
                message = m;
                System.err.println( "It keeps going" );
        }
        public int counter = 1;

        public void run() {
                while( true )    // aka. forever() {
                        System.err.println( message + " [" + counter + "]");
                        counter++;
                }
        }

        public void writeExternal(ObjectOutput o)
                        throws java.io.IOException {
                this.suspend();
                o.writeInt(this.counter);
                o.writeUTF(this.message);
                o.writeUTF(this.getName());
                o.writeInt(this.getPriority());
                o.writeBoolean(this.isAlive());
                o.writeBoolean(this.isDaemon());
                this.resume();
        }

        public void readExternal(ObjectInput o)
                        throws java.io.IOException  {
                this.counter = o.readInt();
                this.message = o.readUTF();
                this.setName(o.readUTF());
                this.setPriority(o.readInt());
                boolean alive = o.readBoolean();
                this.setDaemon(o.readBoolean());
                this.resume();
                this.start();
        }
}
```

Figure 4: Class representing the thread and actor [9]

When the MobileObj is deserialised it updates all the instance variables and executes the `run()` method of LoopThread. This allows the object to be arbitrarily moved without concern for the threads encapsulated within it. The new host (Environment) being decoupled, has no knowledge of the thread contained within.

```
public class MobileObj implements java.io.Serializable {
        public LoopThread loop;

        public MobileObj(String name) {
                loop = new LoopThread();
                loop.setName( name );
                loop.setPriority( Thread.MIN_PRIORITY );
                loop.setDaemon( false );
                loop.start();
        }
}
```

Figure 5:Class of mobile object with thread migration [9]

The problem with this technique arises where the `run()` method has called other methods, which were in turn invoking further methods. Local variables from these methods are not captured. Execution would not resume in this called method. Instead the `run()` method is reinvoked. This will require careful coding techniques so that restarting execution from the `run()` method does not cause inconsistent data.

3 As Strong as Possible (ASAP) Mobility

Strong mobility gives long running threads the ability to continue execution on other nodes. Reasons for doing this were discussed in section 2. However if they use a particular resource on the original node this can cause problems (see previous section). Weak mobility takes a simple approach of ignoring current state (instance variables in Java) and simply migrating and beginning work from the start. It should be noted that this is perfectly acceptable if retention of execution state is a non-issue. An object that, for example, performs local housekeeping operations has no use of state from any previous runs.

Java's serialisation [5] allows the programmer to capture an object, convert it to a bytestream, and move it to a different virtual machine on local or remote nodes. While threads [9] in Java are objects (java.lang.Thread) they are not serialisable. There is no close coupling between threads and the objects from where they were created. Instead the thread executes as a separate entity and uses the class's opcodes for instructions while saving the object's state (instance values) on the heap.

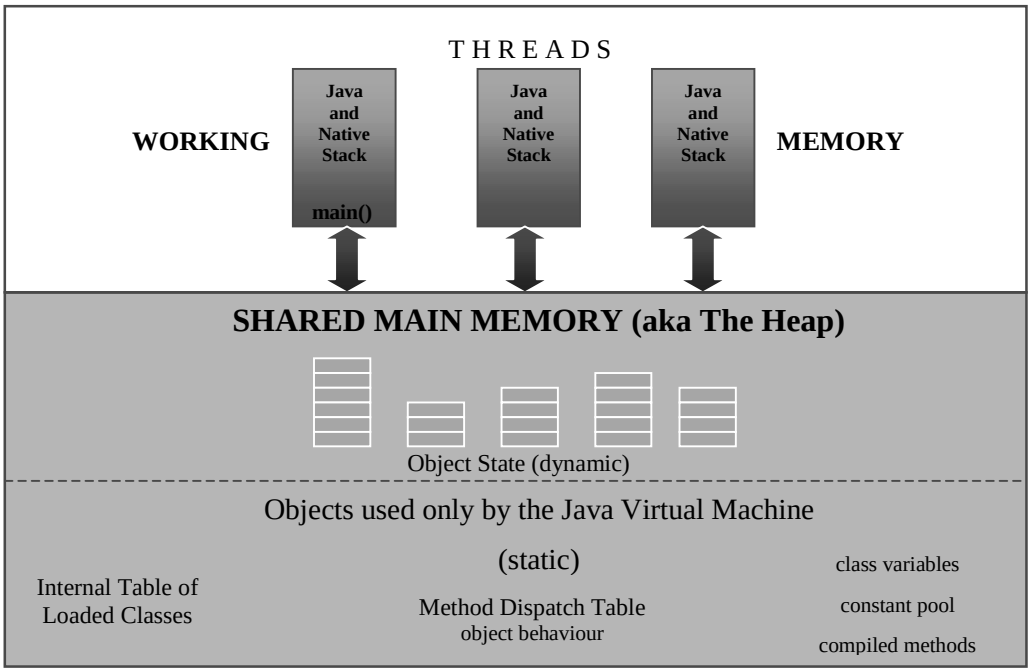

Figure 6: Java Hotspot Virtual Machine memory layout [16]

The scope [5] of state captured by serialisation is limited to non-static and non-transient instance variables (the object state in figure 1). Attempting to serialise certain objects, such as an ORB, will throw a `NotSerializable` exception. These objects need to be declared `transient`. Method's local variables, program counters and other registers (i.e. the thread stack) will not be captured. The state captured is bundled with the object's code and written to an output stream.

3.1 Stack Collapse

Since the thread stack is not captured it may prove prudent to try to collapse the stack naturally. If no methods are being executed when migration is initiated then the stack will be flat (no local variables to be saved), thus allowing the object to be moved and restarted on a new node. Essentially the object is being shut down; no methods are executing. The `main()` method will be the sole remaining frame. The remains of the object's state will be encapsulated in the instance variables. Java's serialisation can capture these variables automatically. It should be noted that the term 'migrated' 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 have occurred.

With the object's flat stack frame and bytecodes on a new virtual machine the object can be re-started. Java does not provide a specific mechanism to do this. Instead, using the reflection API, the programmer can access a pre-determined method name and call it. A simple system such as migrants [10] gives this facility. Essentially it allows an object to complete execution before sending itself to an awaiting host which restarts it by calling a pre-determined method name. This approach is also taken in systems like Voyager [11].

The downside to taking this approach is that prior to migration it is necessary for all threads to finish executing. This will be a large drawback on threads that are executing tasks that can be judged to be isolated from the current host. Clearly forcing a thread that is, for example, calculating a large matrix multiplication, to stop executing to allow a migration appears to be an unnecessary operation. Such a thread should be able to enjoy the benefits of a strong migration. Capturing this thread's entire state (including program counter) and moving it to a different node will have no adverse effects on the application. Such an isolated thread is impervious to changes in its location. Since no references to other objects exist it should be able to continue on a different node exactly where it left off when it was halted.

We require a system which provides As Strong As Possible (ASAP) mobility which uses a combination of data space management and thread state capture so that objects and thread can migrate in a manner which has not been explored yet. The ultimate goal is to provide a mechanism for ASAP mobility where an object will be migrated using strong mobility techniques where possible and using rebinding mechanisms (see below) when it is not possible to simply 'grab' a thread's state. Data space management will ensure that threads that need to retain bindings to particular resources are offered equivalent replacements at the destination site. This seeks to reduce to zero the number of bindings to other remote nodes which, as discussed earlier, is the downfall of existing strong mobility techniques when run on large distant networks.

3.2 Data Space Management

Data Space Management can be summed up by the term 'automatic rebinding'. Prior to migration an object may hold references to many other different services and resources. Objects have generic requirements that are required wherever they may execute. Each host system will provide generic resources that can be automatically rebound to. Examples include display screens for reporting progress, printers, proxy servers, class loaders, replicated databases and object request brokers. An object that migrates can use these resources instead of attempting to use the same resources on its previous home. This allows an object to migrate and continue executing using identical services.. The effect of this is to reduce the dependency the object has on remote resources which can be unreliable. The objective is to only retain bindings to remote resources when they are either essential or desirable.

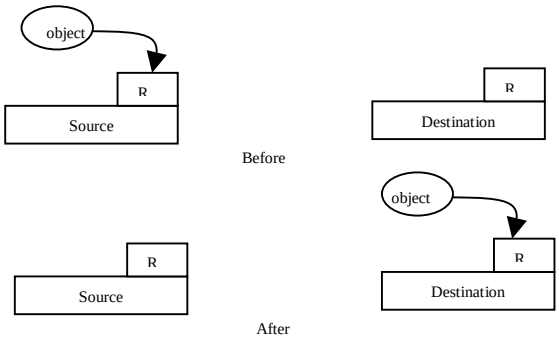

Figure 7: Rebinding

Tending to object's resource needs in this manner is referred to as data space management [2]. When this is done automatically it provides a sense of location transparency to the object. However all possible resources available to an object are not all alike. Some are more unique than others. Fuggetta [2] describe an abstraction mechanism that allows bindings to be managed in a logical manner. Bindings are classified by their relative *strength*.

A binding by '*identifier'* is the strongest type. In this case the resource, which is bound to, is unique and cannot be replicated. Such bindings remain throughout an object's lifetime therefore any migration by the object will retain a network reference to the original resource unless the resource in question is transferable. In this case the resource could also be moved but bindings from other objects would need to be subsequently updated. Such bindings should be avoided, as they need to remain intact over an object's travel thus creating potentially unstable elements with the object.

A binding by 'value' is weaker than by identifier. Such bindings declare that, at a given moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration. This kind of binding is usually exploited when an object is interested in the contents of a resource and wants to be able to access them locally. In this instance the resource could be moved with the object if it is a transferable resource or bound by network reference if the resource is fixed.

A binding by 'type' is the weakest or the types. Such resources are generic resources that are typically available at any node. Examples include printers and displays. This type of binding is re-bound to the local resource on the new node. The management of such resources will be subject to the rules of the architecture. Binding types can increase in strength but never decrease. For example, if an object changes a database resource then that database has now gone from a duplicated resource to a unique resource and so the binding would need to be changed to an identifier type. This represents the resource's new uniqueness.

The FarGo [21] system implements these abstract mechanisms. Pull references ensure that if object A relocates then object B moves also. Likewise, stamp references force object A to rebind to a similarly typed object if it moves to a new location.

An object migrating to another host without concern for its data space separates the algorithm from its location. Achieving this requires a system that can provide resources transparently using a combination of services including naming, trading and relationship. At one level of abstraction each of these services will simply be managing binding. The benefit of abstracting a number of behaviours and managing them by re-binding, is that the algorithm can retain consistent access to a resource whose reference is managed by some external services. So the need for the programmer to explicitly cater for rebinding is now devolved to an external service. It is worth noting that the service can apply policies to the binding process to enforce a user or enterprise requirement such as security or performance policies.

The binding strengths also highlight the problem of resource relocation and how that should be managed. The following section presents a brief overview of an architecture for managing such systems.

3.2.1 Managing distributed applications

The structure of a generic migratory distributed application is displayed in Figure 8. It shows how each component can exist on a different node and yet remain in communication with other components of the application. Here the term node refers to a single processor, typically a single host. The circles represent components, while the dashed lines represent the various network references, or simply links, between them. The components are capable of being migrated to another node yet still

carry on its computation, and more importantly its role in the computation. This is the goal for a real world system but prior to building such a system it is necessary to describe a disciplined abstract structure.

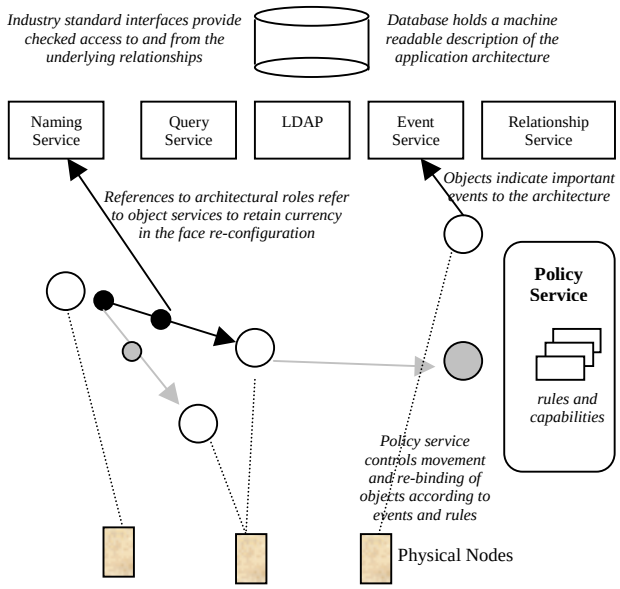

Figure 8: Logical Database abstraction

The complete structure of the architecture can be encapsulated in a logical database. It represents a rational method to store such a configuration because it is well structured, secure and retains data integrity automatically. The logical database can be implemented as a single, distributed or federated database, and we use the central database term solely to convey the methodology to be used in implementing such a system.

Object oriented databases allow for classes and objects to be saved within such a structure. Object-oriented database systems represent some of the most promising ways of meeting the demands of the most advanced applications, in those situations where conventional systems have proved inadequate. These systems when implemented as a distributed system allow the problem of binding reconfiguration to be handled in an elegant manner.

When an object migrates essentially a clone of the object is migrated. Disposal of objects in Java is automatic. For this [5] to occur it is necessary that the object be no longer referenced.

Objects that are to be migrated must first be put through a pre-migration stage. This involves nullifying all references to resources that have remote equivalents. These bindings will be re-established on the destination node..

CORBA Interoperable Object References (IOR) serve as a tested and reliable method of uniquely identifying all objects. Using IOR's as identification for *identifier* (strong) bindings still requires re-binding at a foreign host back to the originating site since the IOR is used as a guide by the ORB runtime in establishing a reference to the actual object desired.

When binding [2] are established by *value* it is required that, at any given moment, the resource must be compliant with a given type and its value can not change as a consequence of migration. Such resources are usually exploited when an object is interested in its contents.

Objects hold vectors containing the object's references for objects it accesses. There is one vector for each of the strengths of binding. All binding information is saved and sent with the object.

Each host has migration and binding service modules that allow the object to move and service its data space respectively. The migration units accept the object and reinitialises it. Prior to resuming operation the binding module uses the objects binding vectors to re-establish connections so that the object can resume operation safe in the knowledge that its bindings are valid and accessible. This is akin to updating the database with new entries.

The *identifier* bindings are re-established via the ORB using the original IORs. Since *type* bindings indicate a generic resource or service these can be found using a relationship service and narrowed using the name service. This data is stored in

the database and accesses using the stateless filters provided.

While data space management is concerned with handling the object's bindings a different process is examining any thread's references to ascertain whether it is capable of being strongly migrated. If a thread does not reference other data outside the migrating object then it is possible, using the JIT compiler API to access the thread's execution environment.

3.3 Strong Mobility in Java

In Java, threads and the objects that created them are separate entities. A Java thread can be created from object's methods but it 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.

The standard Java API does not provide mechanisms to achieve strong migration. There is no way to access these thread objects directly. We investigate whether this is feasible using the Just In Time (JIT) Compiler API. This API gives hooks into the virtual machine that can be used, amongst other things, to extract the execution environment for particular threads.

A University of Maryland project to create network-aware mobile programs resulted in the creation of Sumatra [17]. The design philosophy of Sumatra was to provide the mechanisms to build adaptive mobile programs. Two programming abstractions are introduced, object-groups (OG) and execution engines. Objects are added or removed from object groups, which are then treated as the unit of mobility. OGs move between execution engines. Upon an OGs 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 retention of references as proxies it contravenes our understanding of achieving a reliable strong mobility mechanism.

A single [13] Java stack frame is comprised of 3 parts. The operand stack, the execution environment and the local variable array.

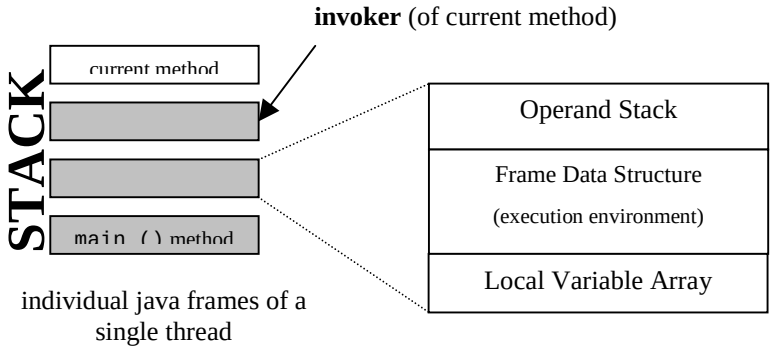

Figure 9: Java Stack and Frame Detail [16]

The operand stack is a 32-bit LILO data structure where values are pushed onto and popped off during expression evaluations. This section is used when invoking methods. When [16] a method invocation expression is evaluated, the target method reference and method parameters are pushed onto the operand stack. Before control is transferred to the method invoked, these values are popped off the top of the operand stack and copied to the local variable array in the new activation frame. The local variable array holds the following variables:

- Value of the `this` keyword
- Method parameters
- Constructor parameters
- Exception handler parameters
- Local variables

The local variable array and operand stack size of each method is determined at compile time and stored in the method's `Code` section as `max_locals` and `max_stack` respectively.

The execution environment (also referred to as the frame data structure) is a fixed sized structure that records the program

counter and holds pointers to the operand stack and the local variable array. In addition it retains the reference to the previous frame data structure and the constant pool for the current class. It is essentially the bookkeeping section of an activation frame and as its size is fixed the JVM can allocate memory for all three sections of an activation frame upon method invocation.

The constant pool [13], generated by the Java compiler, is a variable length structure that cannot be indexed. As the Java Virtual machine consumes the bytecode of an incoming class it parses the constant pool into an array type structure, effectively separating the tagged items so they can be indexed by the operands of a machine instruction as well as other items in the class file. The constant pool is what makes the class file so compact. As a compiler generates a class file it continually searches the constant pool for matching values before adding a new entry. In the end, for any given tag no two entries have the same value. Therefore, a program may reference `java.lang.Object` many times but it will only occur once in the constant pool.

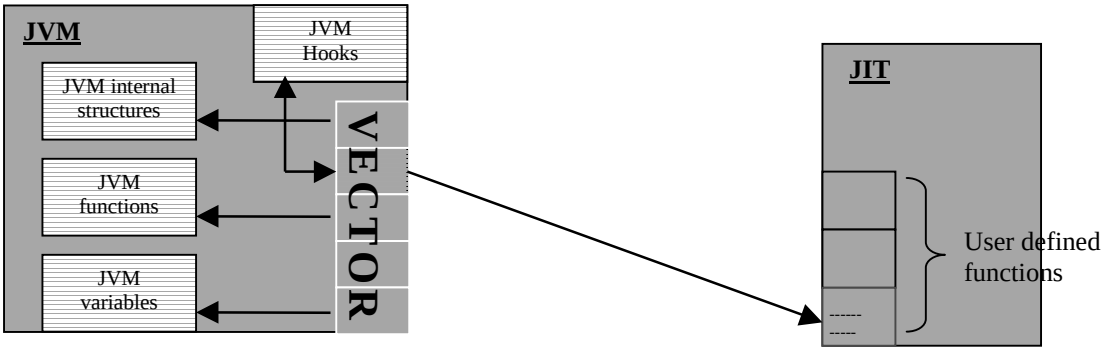

Figure 10: JVM / JIT interaction

The Just-In-Time (JIT) compiler API is provided so programmers can write native code generators and other utilities that can run inside the Virtual Machine. The three types of native code generators available are:

- Ahead of Time recompilers – This rewrites the class file into a 'fat' class file that also contains the native code of one of more platforms. Thus this class file can accommodate several machine architectures simultaneously.
- Ahead of Time compilers – In this case a 'fat' class is once again created but the difference being that it is created from the original Java source file instead of bytecodes
- Just in Time compilers – This compiler runs concurrently with the virtual machine. It recompiles all bytecode into native code for the particular platform. More advanced versions, such as the Hotspot compiler in Java2, identify methods which are used most often and only convert these. It is based on the rule of thumb that 80% of execution time is spent on 20% of the code. Therefore it does not necessarily make sense to compile everything to native code, as it is an expensive operation.

Due to the nature of compilers and recompilers, the JIT API provides access to important inner sections of the virtual machine. Thus, and it is actively encouraged by Javasoft, the JIT API can be extended and used for other purposes.

Using the JIT API involves writing a module or shared library that will be executed in the Virtual Machine (VM). Resembling the paper describing the JIT API we shall refer to all modules which use the API as JIT modules. When started the VM checks if the system properly `java.compiler` has been defined and is not `null`. This property is taken to be the name of the shared library that is then loaded and initialised.

As part of the initialisation process the VM calls the java_lang_Compiler_start function in the shared library. The sole argument to this function is a link vector: an array of addresses of various functions and variables in the VM. Each address in the link vector is a pointer to one of the following:

- Hooks to module functions

The hooks can be used by the compiler to affect the running of the VM. Each entry in the vector is a pointer to a function which are initially set to 0. This way, if the JIT module is never initialised any attempt to call these functions will achieve nothing.

The Java Core API has several methods in the java.Compiler class which, when invoked cause the VM to run an appropriate JIT module method. For example the function `java_lang_Compiler_command()` function in the VM is executed when an application calls the Java method `Compiler.command()`. The definition of the function in the VM is as follows:

```
JHandle* java_lang_Compiler_command (Hjava_lang_Compiler *this, JHandle *x)
{
  if (x == NULL) {
     SignalError(0, JAVAPKG "NullPointerException", 0);
     return NULL;
}
  else if (p_CompilerCommand != NULL) {
     return p_CompilerCommand(x);
}
  else {
     return NULL;
  }
}
```

Figure 11: JVM internal code

When the JIT module is being initialised, the variable `p_CompilerCommand` is set to point to a user defined function in the same module. As this Java method takes and returns an object it could be used for nearly any possible command.

- Important JVM functions

These are useful functions that are used by the VM itself. These range from VM memory allocation (`sysMalloc(), sysFree()`), to method invocation (`invokeJavaMethod(), invokeAbstractMethod()`), to stack manipulation (`CreateNewJavaStack(), GetClassConstantClassName()`).

- JVM variables

These variables control key aspects to the working of the VM. These include the maximum stack size for a thread and flags to prevent the JVM inserting lossy (those which lose knowledge of the original command) opcodes.

- JVM structures

These variables contain pointers to the JVM internal structures representing important classes and interfaces. Examples include `classJavaLangClass` and `interfaceJavaLandCloneable`.

Of interest to us is the `struct execenv *EE()` method which returns a pointer to the current thread's execution environment. The execution environment for a thread can be used as a starting point from where a thread's stack can be analysed. Class files contain information pertaining to the structure of the stack's frames, such as the size of each method's local variable array and stack size. Knowledge of the structures allows a stack scan to look for references to other foreign objects. If the thread has references it must be allowed to complete, thus collapsing the stack upon which the object's data can be serialised normally.

A thread whose stack scan reveals no outside references is suspended. The entire frame can be scanned, saved to be sent separately and re-seeded in a new Virtual machine where it can continue execution from its original checkpoint. This is permitted by the `CreateNewJavaStack(ExecEnv *ee, JavaStack *previous_stack)` function in the JIT API. Using the stack scan from the source virtual machine a new stack can be rebuilt until it returns to its original form. As mentioned already, the class file contains essential information on the form of the stack. Essentially it gives a template of the original structure which needs to have its data refreshed. Depending on memory usage on the new virtual machine, it may be necessary to place the thread's stack into a different section of memory. If such an operation is necessary the original stack data will need to have certain elements changed (i.e. the frame data structure which saves the JVM's stack and program counter registers). This prevents the program from getting 'lost' in memory.

4 Realisation

The mechanism for migration is simply a means of moving objects and threads from one site to another. This takes place in a mutually distrustful environment. Cardelli [7] highlights wide area network postulates that capture the main characteristics of the real world. They include:

1. Separate locations exist.
2. Since different locations have different properties, both people and programs will want to move between them.
3. Barriers to mobility will be erected to preserve certain properties of certain locations.
4. Some people and some programs will still want to cross those barriers.

These points stress that barriers will inevitably exist and mobile code will have to cross them.

The observables [7] that characterise wide area networks (i.e. Internet) have the following implications:

- Distinct virtual locations:
  - Are observed because of the existence of the distinct administrative domains, which are produced by the inevitable existence of attackers
  - Preclude the unfettered execution of actions across domains. Preclude continuous connectivity.
  - Require a security model
- Distinct physical locations:
  - Are observed, over large distances, because of the inevitable latency limit given by the speed of light.
  - Preclude instantaneous action at a distance
  - Require a mobility model
- Bandwidth fluctuations (including hidden failures);
  - Are observed because of the inevitable exercise of free will by network users, both in terms of communication and movement.
  - Preclude reliance on response time.
  - Require an asynchronous communication model.

The migration technique presented above acknowledges and addresses the problems associated with bandwidth fluctuations and distinct physical locations

We address the distinct virtual location dilemma using a policy service. On an abstract level, policies represent a capacity to do something. In the instance of mobile entities the policy will relate directly to the ability or permission to migrate and then, on a more detailed level, the ability to access and use particular resources.

4.1 Life Cycle of a mobile object

Prior to migration, the object must request permission to visit the destination host. A request for residency will also contain a list of resources it requires to access it. Policies enforced on the destination will decide firstly on whether the original migration is permitted and secondly on which resources it will and will not be allowed access.

If the migration requirements are satisfactory the object needs to go to a ‘migration’ mode where its associated threadgroup’s threads are analysed. Those which reference other objects outside itself need to stop executing and return to their `main()`method which will facilitate a serialisation. Threads that have no references outside this object’s scope are suspended and then handled outside the Java runtime. The JIT API module inside the JVM will then take a capture the

thread's stack. Issues regarding synchronisation of their various shutdown and suspend states need to be rigorously examined in this case. Failure to realise this will render any such system useless.

After all these processes have been carried out all data will be shipped to the destination host where they are placed in a Virtual machine to be restarted. Bindings to similar resources are re-established and threads are resumed.

5 Conclusions

We have established that weak mobility semantics do not give the level of service required for system programmed distributed applications on a wide area network. Similarly strong mobility is not feasible because it is assumed that they are running on closely coupled systems. The activation record splitting approach taken by existing systems will not work over large networks / distances.

We proposed a system that offers a different perspective on the movement of an object and any threads that belong with it. The key features are a two-tier approach to migration in an effort to minimise references to remote hosts. No references to foreign hosts will allow threads to run with performance akin to static code. A combination of strongly migrating certain threads and managing data space will give an efficient basis for enterprise systems for use in key domains.

The Java JIT API has not been used for such a task before. We investigate the practicality of taking such an approach.

The data space area has been addressed and work is continuing on perfecting thread migration especially when thread synchronisation is involved.

Future potential work will involve the analysis of the quality of service on the link and then ascertain whether retaining a reference will not be detrimental to the overall operation.

REFERENCES:

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

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

[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] Niranjan Suir, J. Bradshaw, M. Breedy, A. Ditzel, G. Hill, B. Pouliot, D. Smith, "NOMADS: Toward an Environment for Strong and Safe Agent Mobility"

[5] Ken Arnold, James Gosling, "The Java Programming Language", 2nd edition, Addison-Wesley, 1998

[6] Goralski W., Waclawski, D., "Virtual Private Networks : Achieving Secure Internet Commerce and Enterprisewide Communications", Computer Technology Research Corporation, April 1999.

[7] Cardelli, L, "Abstractions for Mobile Computation", Secure Internet Programming: Security Issues for Mobile and Distributed Objects. Lecture Notes in Computer Science, Vol. 1603, Springer, 1999. pp. 51-94

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

[9] Jeff Nelson, "Programming Mobile Objects with Java", Wiley Computer Publishing, 1999.

[10] http://www.cs.tcd.ie/Simon.Dobson/migrants/

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

[12] Yellin, F, "The JIT Compiler API", Javasoft '96. *available from* java.sun.com/docs/jit_interface.html

[13] Lindholm, T., Yellin F., "The Java Virtual Machine Specification, Second Edition" Addison-Wesley, 1998.

[14] Ophir Holder, Israel Ben-Shaul, Hovav Gazit "System Support for Dynamic Layout of Distributed Applications ", Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pp 163-173, May 99, Los Angeles, USA

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

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

[17] 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

[18] 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

[19] 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.

[20] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns Elements of Reusable Object-Oriented Software", Addison Wesley, October 1994.

[21] 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.