

# Thread Persistence and Migration

Doctoral Dissertation Proposal

Wei Tao

Department of Computer Science

University of Utah

Email: tao@cs.utah.edu

## 1 Motivation

Threads are a well-established programming abstraction known as *lightweight processes* [15]. Unlike operating system processes, multiple threads execute within a common address space, permitting access of shared resources. Therefore, threads require significantly fewer resources, but need synchronization mechanisms for coordination and communicating shared objects.

The motivation of this research is to design a programming methodology and tools that enable a program in multithreading programming languages, such as Java [10], to externalize its threads' execution states, and to apply the technique to support thread persistence and mobility at language level.

### 1.1 Persistent Threads

A persistent program is a program written in a language that permits the creation and manipulation of objects independent of their life times. Thread persistence is an important feature for the following reasons. First, thread persistence is a necessity for an orthogonally persistent programming system, in which the manner that data is manipulated is independent of its persistence mechanism. An absence of thread persistence violates orthogonal persistence. Programmers end up coding translations to and from data structures that can be made persistent. This distracts the application programmer, clutters the code and introduces errors associated with inconsistent translations. Secondly, the lifetime of a traditional thread is limited to that of its enclosing operating system process. In order to conveniently control long-term activities, such as business workflow, and map them to threads, it is desirable to be able to store threads persistently.

A number of ongoing projects in the industry offer Java persistence at the object level, including Object Design's PSE/PSE Pro [21], and Sun Microsystem's PJama [14]. But thread persistence is a missing functionality of all those systems. More discussion on this omission is provided in Section 3.

## 1.2 Mobile Threads

Mobility and persistence are intrinsically related because mobility is conceptually similar to persistence if the store is replaced by the runtime environments of another program. The same procedures for marshaling and un-marshaling arguments to remote procedures can be used to export objects to a file or import objects from a file. The same algorithms can be applied to both program-store and program-program communication. A mobile computing system usually needs persistence to achieve computation reliability and fault tolerance.

Mobile threads can be seen as a refinement of mobile computation. A computation is code plus the context of its execution. Mobile computation [4] requires interrupting execution at an arbitrary point, moving the state of a runtime system from one site to another and then resuming execution. Traditional languages do not support mobile computation and are not well suited for network computing. Some modern programming languages, such as Java, support multithreading and network capabilities. However, a language needs a mechanism to capture execution states during runtime to fully support mobile computation. Most languages including Java do not provide an explicit way to capture states of their partial computations.

A mobile agent is a kind of mobile computation that can migrate under its own control from host to host in a heterogeneous network. Using agents may reduce network traffic because they eliminate intermediate data transfer by moving to the network location of the user or resource. Moreover, agents can travel into the network and act autonomously even if their creator has been disconnected. This makes agents suitable to be used in systems which have unreliable or non-permanent network connections, such as mobile computers.

There are various ways to implement mobile agents. They can be treated as a single thread of control or as a collection of synchronized mobile threads. The Java mobile agent systems, such as Aglets [13] and Odyssey [8], depend on Java serialization [25] to pack the migrating agent objects and do not transfer control state information with them. Hence agents in such systems must begin as new Java executions upon arrival. In order to logically restart the agent in the state it held before its relocation, an agent programmer must ensure that all accessible data is saved before migration. Furthermore, an explicit note of which task the agent was running must be made and then the task must be restored to the same computational state after migration. In Odyssey, a task list is used to specify one task for each destination. The programming style are restricted and a more strict structure is required. For example, because an agent cannot be moved in a middle of a recursive function, the program has to be reorganized and one method may need to be divided into several methods. For some algorithms and applications, it is difficult and awkward to program with this restriction. This is certainly a limitation of Java agent programming.

## 2 Problem Statement

A thread describes a single sequential flow of control in a program. Having multiple threads in a program means that at any instant the program has multiple points of execution, one

in each of its threads. A multithreaded programming language, such as Java, supports multithreading at the language level. It provides thread control and synchronization to coordinate concurrent threads that execute different code against shared data. For instance, each Java thread has its own stack space, but all threads share the heap with support of synchronizations.

In order to save or migrate a thread, the state and resources of the thread need to be preserved. In addition, in order to resume execution of the same thread, the execution stop point of the thread is also needed. However, a thread object in most languages cannot be passed around and manipulated with its execution state. In order to save or migrate thread objects, we have to find a way to collect stack information and save the execution point of each thread. Furthermore, because threads share some common resources, a synchronization scheme is needed when saving and resuming threads.

In this research, the Java programming language is used to experiment with an execution state externalization technique as well as thread persistence and migration. Java was designed for mobile code on heterogeneous platforms on the Internet. An architecture-independent representation of program code, Java bytecode, is shipped over the network and interpreted remotely. To preserve Java's portability feature, the system should not modify Java compiler or JVM [17]. In other words, the model should be applicable to any Java compiler or JVM.

### 3 Related Works

This research relates to several active research areas. Persistent programming languages are one of them. Java is not intrinsically a persistent language, but it can be extended to support persistence. Much research is going on to make Java a persistence-capable programming environment. It is also worth mentioning some traditional persistent programming languages, which introduce some interesting strategies and features, especially their support of persistent threads.

Another area is mobile computation and mobile agents. There are languages that are designed for the purpose of mobile computation. General Magic's Telescript [29] and DEC's Obliq [3] are two pioneers in the area. Because of Java's dynamic nature and advanced networking capabilities, many Java mobile systems have been developed. However, Java is not well suited to support mobile computation compared to the languages mentioned above.

#### 3.1 Persistent Programming Languages

There are programming languages designed to provide a full integration of persistence. Napier88 [20] is a research persistent programming language that support orthogonal persistence [2]. It supports abstract data types and parametric types, including parametric procedures. Type equivalence is structural based on a set of base types and a set of constructors, and the type of an object can be obtained at run-time. Procedures are first-class objects and can be made persistent to survive program execution. Napier88 supports dy-

dynamic binding and type-checking, memory and disk garbage collection, type-safe linguistic reflection, and heterogeneous architecture. However, it does not directly support persistent threads.

Tycoon [18], on the other hand, provide functionality to make thread objects persistent. It is designed to offer a persistent programming environment for data-intensive applications in open environment. Tycoon is a persistent, polymorphic, higher-order functional language extended with imperative features. All language entities in Tycoon have first class status and can be manipulated as standard data. These data can be stored persistently and are transparently managed by the Tycoon environment. Persistent threads [19] in Tycoon are treated as a novel form of bindings from data in persistent object stores to activated code. These concepts can be further extended to introduce thread migration, where threads may access remote or mobile resources in heterogeneous networks.

### 3.2 Java Persistent Systems

Much research is underway to make Java persistence capable. However, none of this research seems oriented toward persistent threads. The main difficulties are how to access stack information and how to deal with synchronization of shared resources.

PJama [14] is a prototype implementation of an orthogonally persistent variant of the Java platform. It provides orthogonal persistence with no changes to the Java language and a small modification to JVM. The system also includes flexible transaction models. Persistence of Thread instances is not supported in PJama. A thread can be made persistent in a store, but no actual thread of control will be created when a fresh execution of the virtual machine uses that store. An attempt to invoke operations on the thread may possibly crash the system.

Another Java persistent system is ObjectStore PSE/PSE Pro, which is a lightweight persistent storage engine. It is portable, written in pure Java, and designed to be used in a variety of application settings. It provides a transparent API, random access to persistent data, unique object identity, and atomic transactions. PSE Pro also supports fine grain concurrency. However, Thread cannot be made persistent-capable in a PSE system and therefore cannot be saved in a store.

Howell [12] described a system called *icee* that provides thread persistence by checkpointing a JVM. However, it is inefficient and inflexible because it saves all data in a flat file by comprising a memory image of the checkpointed JVM process. Moreover, its persistent data representation is platform specific.

### 3.3 Mobile Code Languages

Before Java was born, only a few mobile agent systems were under development, based primarily on research languages like Tcl, Scheme, and Obliq. Telescript was the only commercially available mobile agent system initially. Telescript is an object oriented language designed for the development of large distributed applications, such as electronic commerce. Telescript execution units or threads can be migrated while executing. A *Place* represents

virtual space in which objects can interact. An *Agent* is an object that can migrate between places.

Another example is Obliq, an untyped, object based, lexically scoped interpreted language. In Obliq, any value can be transmitted between hosts, including closures and object references. It is prototype-based language, in which objects are created by copying existing objects instead of by instantiating from classes. Obliq is dynamically typed. Type errors are caught cleanly and propagated to the origin site.

### 3.4 Mobile Computation in Java

IBM's Aglets [13], ObjectSpace's Voyager [22], and General Magic's Odyssey [8], are the three leading commercial Java mobile agent systems. However, none of them support transparent mobile computation. Aglets and Voyager both use a callback model based on the Java event delegation model. During its life cycle, an Aglet receives various kinds of events in response to its actions and the corresponding callback methods are invoked. Upon arrival, an Aglet starts its execution from the beginning of its `run()` method. The control state of its previous computation is lost. The programmer needs to record the next task by hand to know which task the Aglet to perform after arriving at the new host. A Voyager agent starts its execution in a method specified in its `moveTo()` method arguments after arriving at its destination. Hence an agent executes different methods at different hosts. Odyssey has a subclass of *Agent* called *Worker* that maintains a set of destination-task pairs specifying the task an agent performs at each destination. The agent can start its new task at its destination, but it has no way to resume its old computation stopped at its old host. In these systems, the code describing a long-term activity has to be fragmented and it is not possible to transmit recursive data structures.

Halls [11] described a Scheme [5] interpreter which converts a program into a number of closures each of which performs a small piece of its computation. At each stage in the program's execution, the program's state and continuation are defined by one of the closures, which can be saved as a sequence of bytes. By running this interpreter on a Java virtual machine, Java bytecodes compiled from Scheme programs can save their states in byte streams at any stage. However, the interpreted code runs much more slowly than code written directly in Java. Furthermore, the interpreter only runs programs written in Scheme.

A more straightforward approach [1, 24] is to modify Java virtual machine implementation to support checkpointing of Java programs. However, this approach negates the ubiquity benefit of standard JVM.

## 4 Proposed Approach

In this section, we propose a technique to extend Java to support persistent threads and mobile computation. The technique does not require a new Java compiler or any modifications to the JVM. Instead, the desired Java extension is achieved by transforming the Java

bytecode, which can be performed either after compile time, or at load time.

#### 4.1 Continuation Semantics of Shutdown and Restart

During execution, a program is evaluated according to its control structure. Under unusual circumstances, a program may exit abnormally. After exiting from its execution, its control state and temporary data disappear. When the program restarts, it has to restart all over again. By contrast, in order to resume a computation without loss of previous computation, the program data and control states must be saved before the program exits.

In a continuation semantics [9], we express the denotation of constructs in terms of continuations, which represent the rest of the program execution. Continuations make the control behavior of a program explicit by representing the portion of the computation to which the result of a subcomputation should be sent. Using continuations, a command statement in a program has the type signature:

$$C: Com \rightarrow Env \rightarrow Cc \rightarrow Cc$$

where  $Env$  has domain

$$r \in Env = Id \rightarrow (Denotable-value + unbound)$$

and  $Cc$  is a function representing command continuations with domain

$$c \in Cc = Store \rightarrow Ans$$

where  $Ans$  is the domain of final answers and  $Store$  is

$$s \in Store = Loc \rightarrow (Storable-value + unused)$$

Similarly, the semantic function for expressions can be represented in the continuation style using expression continuations:

$$E: Exp \rightarrow Env \rightarrow Ec \rightarrow Cc$$

where  $Ec$  is a function representing an expression continuation. Its domain is

$$k \in Ec = Expressible-value \rightarrow Store \rightarrow Ans$$

The expressible value argument to an expression continuation is the intermediate value of the partially evaluated expression.

Since a continuation explicitly represents program control, we can resume a stopped execution if we have saved its continuation and its store at its stop point. When program restarts, the saved continuation can be used to resume execution. Let the domain of  $Ans$  include  $(Cc \times Store)$ . The semantic equation of a shutdown command is simply:

$$C[\textit{shutdown}] r c s = (c, s)$$

which means that the answer of a shutdown command is a pair consisting of a continuation  $c$  and a store  $s$ . Let the semantic function of a program be  $P \in Pro \rightarrow Cc \rightarrow Cc$ , an ordinary start of a program is as  $P[[P]] (\lambda. finish) \phi$ , where the program code  $P$  is evaluated with an empty store  $\phi$  and a continuation which when sent a store stops with final answer  $finish$ . By contrast, the semantic equation of a program when it is restarted is:

$$P[[restart]] c s = c s$$

This takes the saved continuation  $c$  and store  $s$  and continues execution from the saved execution point they represent.

## 4.2 Externalizing Java Execution States

A continuation, being a complex function to denote semantics of control flow, is usually not a structure that can be saved or externalized. In most programming languages including Java, continuations are not first-class and are not storable. To be able to externalize continuations, we introduce a new concept, “storable continuation”, to explicitly represent control states [28].

The sequence of control states can be implemented using a *list* or other similar data structures. A natural choice is a *stack* structure because continuation is a representation of control stack in a sense. In a multithreading programming language, each thread has its own storable continuation separated from other threads. Moreover, during the process of externalizing control states after shutdown, a continuation should be visible in any scope. The semantics of exception handling perfectly meets our purpose to handle control state externalization. A storable continuation can be implemented as a special kind of exception. In effect, stack structure is constructed in the exception arguments to store the control states. When an exception occurs, it is either caught by its handler or propagates to its dynamically surrounding environment. Because each Java thread has a separate stack space and therefore a separate control state, we can save the control state of each thread separately as if it were the only thread. When saving object graphs and shared data, we must take multithreading into consideration.

When saving the control state of a thread, its stack is collapsed and each of its stack frames is recorded in LIFO call chain order. During restart, the saved stack is rebuilt in reverse order according to the saved records. We now describe a mechanism for accomplishing these dual operations.

An object of a new subclass of Java’s `Exception` class, `StoreException`, is thrown by instructions that trigger shutdowns. Class `StoreException` contains a data structure `Record` that is implemented as a LIFO stack. Its declaration is shown in Figure 1. The method `log(Object item)` pushes a new item onto the record stack. The method `Object getNext()` pops the next item from the top of the record stack and returns it. The last method `Record getRecord()` returns the record stack contained in this object.

A try-catch block is inserted around every instruction that may throw a `StoreException` object. There are two kinds of instructions that may do this:

```

public class StoreException extends Exception {
    private Record record;

    public void log(Object item);
    public Object getNext();
    public Record getRecord();
}

```

Figure 1: Class StoreException.

- Instructions that trigger a shutdown;
- Instructions that invoke a method that throws StoreException. When a method contains instructions that trigger shutdown or instructions that invoke a method that throws StoreException, it rethrows StoreException in a manner to be described shortly.

When a StoreException is caught, the thread saves the current activation record in the stack structure contained in StoreException. In addition to parameters and local variables, the position of current instruction is also saved. We count the instructions which may throw StoreException starting from 1 in each method. So, the first instruction in a method that may throw StoreException is labeled as 1, the second is labeled as 2, and so on. After the label is saved in the given StoreException along with the method's activation record, the StoreException is then rethrown.

Figure 2 shows the shutdown code of a method *f* with return type *rtype* and *n* parameters. To facilitate readability, the inserted shutdown code is presented as Java source code rather than bytecode. *s<sub>1</sub>* is the first bytecode instruction in *f* that may throw a StoreException. In this method, there are *k* local variables and they are saved in *se* in reverse declaration order. The instruction's label is saved next. Following the label, values of parameters are saved, again in reverse declaration order.

Rethrowing the exception causes it to be caught by the caller of this method, and so on. The control state and local data of the caller are also packed into the exception in the catch clause in the same manner. The exception is then rethrown. The process continues until execution reaches the outermost level along the call chain of the thread – either a *main()* or a *run()* method. At that point, the data collected in the exception is extracted, and either saved into a persistent store or serialized and saved into a file, to be used when the program restarts.

The restart process is exactly the opposite of the shutdown process. Each method that throws StoreException has a corresponding restart version. The restart version of a method *f* as shown in Figure 2 has a name of *f\_restart* and one more parameter: the saved object of StoreException. Note that we need to make sure that the added method does

```

rtype f(ptype0 p0, ... ptypen pn) {
    ...
    try{
        s1;
    } catch(StoreException se) {
        se.log(lvk); // local variables
        ...
        se.log(lv0);
        se.log(1); // instruction label
        se.log(pn); // parameters
        ...
        se.log(p0);
        throw se;
    }
    ...
}

```

Figure 2: Shutdown code within a single method.

not have the same name and signature as another method name in the same class.<sup>1</sup> The argument values are obtained when the method is invoked: `f_restart(saved_se.getNext(), ..., saved_se.getNext(), saved_se)`.

Within each restart method, a switch instruction is used to dispatch execution according to the saved instruction label. Local variables are initialized by their saved values before executing the instruction with the saved label.

Figure 3 shows method `f_restart`, where  $s_1''$  is the restart version of  $s_1$ , and  $s_1'$  is  $s_1$  with shutdown code inserted as shown in Figure 2. As in Figure 2, the inserted shutdown and restart codes are shown as Java source code although they are actually implemented at the bytecode level. After executing the restart code of the instruction with label `nextInst`, the program jumps to the next switch instruction which dispatches execution to the shutdown code of the instruction following `nextInst`. Then the program continues sequentially in shutdown enabled mode.

When the program restarts, it first fetches the saved log records, if any. After restarting all other saved threads, the main thread resumes its own computation from the point at which it previously stopped. There are two possibilities for the saved instruction: it is either the instruction that triggered the previous shutdown, or it is a method invocation that propagated a `StoreException`. If it is the instruction that initiated the shutdown, the

---

<sup>1</sup>With an extra parameter `StoreException`, it is unlikely that another method with the same name and signature exists in the same class. If this situation does occur, another parameter with a newly created type is added in the new method's parameter list to guarantee its uniqueness and prevent overloading.

```

rtype f_restart(ptype0 p0, ... ptypen pn, StoreException saved_se) {
    int nextInst = saved_se.getNext();
    switch(nextInst){
        case 1 :
            t1 lv1 = saved_se.getNext();
            ...
            tn lvn = saved_se.getNext();
            try{
                s''1; // restart version of s1
            } catch(StoreException se){
                se.log(lvk); // local variables
                ...
                se.log(lv0);
                se.log(1); // instruction label
                se.log(pn); // parameters
                ...
                se.log(p0);
                throw se;
            }
            break;
        case 2 : // the next instruction that throws StoreException
            ... // restore code for case 2
        ... // the rest of restart instructions in this method
    }

    switch(nextInst + 1){
        case 1 :
            ... // instructions before s1
            s'1; // s1 with its shutdown code
        case 2 :
            ... // instructions between s1 and s2
            s'2; // s2 with its shutdown code
        ... // the rest of the code in this method
    }
}

```

Figure 3: Restart method of f.

restarting process for this thread is completed and the thread's activation record has been restored to that it had when shutdown occurred. Otherwise, if it is a method invocation, the restart version of the method is invoked. The process continues until execution reaches the point at which shutdown occurred.

If another shutdown is triggered before restart process is completed, the program aborts without saving its execution state. In this case the previously saved record is used at the program's next restart. As in database recovery operations, restart is an idempotent operation.

### 4.3 Dealing with Synchronization

When shutdown occurs, a thread may be executing an instruction within a synchronized block. During shutdown, it leaves the block and releases the lock on the associated object. If only one thread is shutting down and all other threads keep running, that thread releases all locks it has and stops. All other threads continue as normal. However, if all threads shutdown at once, extra care is needed to synchronize them during restart.

When the program restarts, all threads start to execute their restart code and acquire each new lock when they enter a protected block under that lock. Because it is exactly the opposite process of shutdown, the threads would acquire all those locks they released during shutdown when restart is complete. However, some threads may complete their restart process before other threads. If one of such threads enters a synchronized block in which another thread stopped its previous execution, it would prevent the second thread finishing its restarting process and cause inconsistent state. To solve this problem, we have to make sure that all threads have completed their restarting process before any thread starts a new computation. A naive way of doing it is to set each thread a higher priority when it is restarted. At the point when the restarting process is completed, its priority is set back to its original priority. In this way, the threads doing restart have higher priorities than those already finished restarting. Before a thread starts new computation, a `yield()` is inserted to give other threads a chance to execute. However, it is not reliable to depend on priorities to synchronize the threads. A better approach is to use a centralized monitor thread to synchronize the restarting threads. When a thread finishes its restart process, it notifies the centralized thread and suspends itself. The centralized thread uses a counter to keep track of how many threads have finished restarting. When all threads finished restarting, it notifies them to resume their execution.

### 4.4 Saving Objects Persistently

After the program's execution states are collected into objects, a persistence mechanism is needed to store the state of objects in a non-volatile manner so that when the application shuts down, the objects are not destroyed. There are several options to choose from in order to satisfy different persistent object management needs [23].

Java serialization [25] is a simple yet extensible persistence mechanism for Java. It defines interfaces for writing objects to and reading objects from a stream of bytes. A

default implementation of this interface is provided, which users can optionally customize by writing their own per class implementations. When an object is serialized, its references to other objects are recursively traversed and serialized. As a result, a complete serialized representation of the entire reachable object graph is created. Deserializing an object simply inverts this process. An object is read from a stream of data and restored in the state it held when serialized. If the object refers to other objects, they are recursively deserialized as well. This capability is a very powerful feature, but it suffers from poor performance when applied to a large number of objects. It also lacks undo or abort capabilities. A database management system is needed to provide reliable management of large numbers of persistent objects.

Java serialization is suitable for shipping copies of objects to another JVM. It will be used when migrating thread objects. When checkpointing applications and saving threads persistently, a database management system such as Object Design's PSE Pro [21] is more appropriate.

## 4.5 Implementation Plan

This section presents our implementation plan. We first specify how to achieve code transformation. Then we discuss the use of persistent system and mobile mechanism to support thread persistence and migration.

### 4.5.1 Code Transformation

The code transformation described above will be implemented using Java bytecode rewriting technique. Bytecode transformations [6] can either be performed after compile time or at load time. Class loaders [16] provide an elegant means of extending the JVM with new features without actually modifying it. A custom `ClassLoader` object may be used to intercept the standard procedure of loading a class and perform some value added transformations before actually passing the bytecode to the JVM. However, rewriting classes during class loading introduces performance penalties. Sometimes rewriting a class may need information about other classes, which may not be available at the time. Also, classes cannot be modified after loading, which prevents retroactive post-processing. These and other considerations may make transformations impossible to do on-line in practice. On the other hand, in some environments such as downloading class files dynamically through a network, it is necessary to rewrite classes on-the-fly. We plan to use a bytecode rewriting tool, *JavaClass*, developed by M. Dahm [7], to implement the code transformation.

We transform programs at the bytecode rather than source code level for several reasons. First, Java source code is not always available for downloaded applets and applications. Secondly, code transformation at the bytecode level is simpler to implement, while achieving the same functionality as that at the source code level. This is because Java bytecode comprises a small and easy-to-understand set of instructions deployed by Java compilers in a small number of relatively simple idioms. For example, bytecode within a method is a linear sequence of instructions with all the method's lexical blocks compiled away, thereby

simplifying control states. Finally, bytecode has finer granularity than that at the source code level. An instruction at the bytecode level contains one atomic operation, facilitating precise placement of inserted code.

#### 4.5.2 Persistent Threads

The code transformation technique collects the execution states of threads. In order to make thread states outlive the context of their capture, they need to be stored persistently to be resumed later.

We will use Object Design's PSE Pro to store application objects including thread objects. In PSE Pro, threads are not persistent-capable because it does not make sense to store a thread without capturing its state information. Our system will enhance the persistence feature by using a language transformation technique. The thread execution can be stopped and saved in a persistent store and retrieved when it is restarted.

There are two ways to trigger shutdown. In the first way, the user can send a shutdown signal at an arbitrary time to stop the execution [27]. A shutdown manager is created at the beginning of each application program as a daemon thread. It waits for shutdown signal from user. When a shutdown signal is received, it interrupts all user threads to inform them to collapse their control stacks and save their execution states. After all user threads are stopped, the shutdown manager saves the log information into a persistent store which can be retrieved later. Because Thread objects are not savable in PSE Pro, the object of a class that implements a Runnable interface is saved instead. When the program restarts, new thread objects are created from the saved Runnable objects. The Runnable objects are put into a hashtable with a unique key. Their saved log records are retrieved by the key when program restarts. Some preliminary experiments and performance measurements can be found in [27].

The second way is to explicitly specify shutdown points in Java programs. Method `ShutdownManager.stablize()` is used to shutdown the whole program explicitly. An individual thread can also stop and pack its state during execution if it is an instance of a class that extends `SSThread` class instead of `Thread` class. `SSThread` is a subclass of `Thread` in which a method `saveState` is implemented to stop the thread control by throwing a `StoreException`.

#### 4.5.3 Thread Migration and Mobile Computation

Thread objects can also be shipped to another machine and resume remotely. Because the execution environment changes at the remote site, we need to decide what kind of information needs to be shipped and what kind of information needs rebinding. This problem actually deals with object identity and reference sharing. The state saving technique will be applied to existing Java mobile agent systems and the above problem will be solved based on the model used in the selected mobile system.

Our current plan is to use ObjectSpace's Voyager as a framework to implement our Java mobile agent system that supports saving execution states. We choose Voyager rather than other Java mobile agent systems for the following reasons. First, Voyager is a mobile agent

system as well as a mobile object system, which provides more flexibility and functionality. It treats agent as a special kind of object that can move independently but otherwise behave exactly like any other object. Every objects can send messages to an agent even as the agent is moving. In addition, Voyager allows a user remote-enable a Java class without modifying the class source in any way. Secondly, Voyager includes a rich set of services for transparent distributed persistence, scalable group communication, and basic directory services. These services make the system easier to be extended to support more varieties of applications. Finally, ObjectSpace continuously adds richer features into Voyager and uses Voyager core technology to develop their own new products. For example, Voyager can use PSE/PSE Pro as its database system and it is compatible with both Netscape Navigator 4.0 and Internet Explorer 4.0. These compatibilities make it very attractive for developing Internet applications such as e-commerce.

Like agents in other Java mobile agent systems, a Voyager agent cannot really resume execution from where it stopped after arriving to the new destinations. In order to avoid Java's inability to maintain an execution stack across virtual machine boundaries, Voyager uses callback style of programming. An agent can move to another program and continue to execute when it arrives by sending itself `moveTo()` with the address of the destination program and the name of the member function that should be executed on arrival. Note that the agent does not resume its execution. Instead, it starts its execution from a new method. Because an agent is deactivated conceptually when it executes `moveTo()`, a programming error occurs if any code other than exception-handling code follows these methods. Therefore, an agent is programmed in a way that one task at a site is organized in one or more methods, but two tasks can not be put into one method. In other words, a agent can not be shipped to another site in the middle of a method execution.

We will integrate our state saving technique with Voyager to enable a more flexible programming style and a conceptually "real" mobile computation. A special kind of agent, `SSAgent`, will be defined as a subclass of `Agent`. It has two new methods: `saveState` which stops execution and starts to pack state information, and `ssMoveTo` which is a state saving counterpart of the ordinary `moveTo`. Method `saveState` takes no arguments. In the method body, a `StoreException` is thrown, so a try-catch block will be inserted around every invocation of `saveState`. The `StoreException` is propagated up along the dynamic call chain until it reaches the top level. All methods contain invocations to `saveState` can potentially throw `StoreException` and try-catch blocks need to be inserted around their invocations. In the catch clauses of `StoreException`, the local variable values and control information are stored in the exception as described in Section 4. Instead of `main` or `run`, the top level of a Voyager agent is the start method upon its arrival to a new destination. An instance variable `topMethod` is included in `SSAgent` to record the current top level method name. It is updated whenever an agent moves. At the top level, the collected state information is extracted from `StoreException` and saved as an instance variable of the agent. Method `ssMoveTo` has one parameter, the destination the agent needs to go. It is treated the same as `saveState` until after collecting the execution state. After execution state is put into an object, the agent calls `moveTo` with arguments of the destination specified in `ssMoveTo` and

the name of the restart version of the current top level method indicating a starting method upon arrival. Each method that may throw `StoreException` has a restart version as described in Section 4.

An alternative plan is to use IBM's Aglets as the experimental mobile agent system, which uses Java Serialization to store and ship agents. Aglets migrate between computing locations called hosts, which are responsible for providing the resources agents need to work and for handling the mechanics of packaging an agent and moving its resource and data pieces from one host to another. The aglet has a `run()` method, which represents the entry point for the aglet's main thread. This is similar to the `main()` method of a Java application, except that `run()` is invoked each time an aglet arrives at a new aglet host. This is due to the inability of aglets to transmit the state of their execution stacks. To enable an aglet to take its control state, we apply our code transformation technique to the aglet code. Before it is dispatched to another host or saved on a disk, every threads belong to the aglet collapses its control stack and collects each stack frame including current execution point. Then they are saved into a hashtable structure as an object variable in the aglet. After an aglet arrives to a new host or restored from disk, the saved hashtable structure is used to guide the restart process of the aglet.

If the above plans does not meet our expectations, we will implement our own mobile agent system using Java RMI. When an agent is ready to be migrated, it calls method `pack()` that throws `StoreException`. The exception is then propagated to the top level with all state information packed in it. After the state information is saved as a instance variable, the thread object is migrated to its next host by calling `go(destination)`.

## 5 Applications

The ability to save program states into one simple object is very useful for multiple applications. This section discusses some possible use of this technique and presents several sample applications in detail.

### 5.1 Logging

For some applications, especially those with long-term computations or complicated multi-threading executions, it is convenient if partially finished computations can be saved persistently. If the computer has to be turned off or some input data is not available at the time, the whole execution can be freed and saved in a store. It can be reactivated whenever it is ready to be resumed.

In addition to preventing lose of partial computations, state saving also provides a way to backtrack to a previous context. A history of the execution states can be maintained so that a program may start from different points under different conditions. This is particularly useful if a user wants to try different inputs at different points of a program. For example, a web searcher may want to enter different refinement queries to a information retrieval interface in a previous context; a board game player may want to go back to the previous

setting and try different steps.

## 5.2 Moving States

Being able to move execution states to different environments makes it possible and convenient for varieties of applications. For example, keeping all states in one object allows a program to be easily moved among servers. This is useful if a user has to rent time on servers and cheaper alternative is found, or if he moves and wants to use a server local to his new location. The state saving technique can also be used for delivering software in fast starting format.

A mobile agent without the ability to capture control states has to explicitly record what to do when it restarts execution at its destination. The code for a task has to be fragmented in order to be executed individually at different destinations. In addition, the agent execution has no way to suspend itself, save it in a local database, and resume execution later. The extended mobile agent system described in Section 4.5 overcomes the above limitations. It provides transparent mobile computation by being able to save the entire execution states of the agent including the control stack.

## 5.3 Examples

This subsection contains some simple real-life business applications that will be implemented using our state saving persistence and mobile agent system described in Section 4.5.

### 5.3.1 Discrete State Simulation (DSS)

The first application example is a simulation of scheduled concurrent events. A discrete state simulation system is used with different examples for different purposes. In this system, multiple threads are scheduled synchronously to simulate concurrent events.

The DSS code as well as its example applications will be processed using our post-processor. After the class files are processed, the program starts in a JVM. A shutdown manager waits for signals from user. If a shutdown signal is received, the shutdown manager interrupts all threads to inform them to begin collapsing their stack frames. At the top level of each thread, the information collected is extracted from `StoreException` and put into a hashtable which is then stored persistently in a PSE store. When the program restarts, a shutdown manager is created first. It then checks if there is a store specified by the user. If not, the program starts afresh. Otherwise, the program fetches its restart record and restarts accordingly.

### 5.3.2 Servlets

A servlet [26] is a Java module that can be loaded into and runs inside a network server, such as a web server. It receives and responds to requests from clients. Initializing a servlet involves doing any expensive one-time setup, such as loading configuration data from files or starting helper threads. Destroying a servlet involves undoing any initialization work

and synchronizing persistent state with the current in-memory state. After the server loads and initializes the servlet, the servlet is able to handle client requests. It processes them in its `service` method. Each client's request has its call to the `service` method run in its own servlet thread: the method receives the client's request, and sends the client its response. Servlets can run multiple `service` methods at a time.

A servlet with potentially long-running `service` requests should keep track of how many `service` methods are currently running. Its long-running methods should periodically poll to make sure that they should continue to run. If the servlet is being destroyed, then the long-running method should stop working, clean up if necessary, and return. In our program, we will rewrite `service` method and any auxiliary methods they call by inserting `throw new StoreException()` after poll code. The state information is collected and the exception is propagated up until it reaches the top level: the `service` method. Then the state information is extracted from the exception and saved persistently.

### **5.3.3 A Workflow Management System**

Workflow management deals with the specification and execution of business process. It allows one to dynamically define, execute, manage, and modify business processes. Mobile agents are particularly suitable for workflow applications because, in addition to mobility, they provide a degree of autonomy to the workflow item. Each business process can be handled by an agent. The agent follows the steps in a previously defined process, migrates to each site and gathers information. While traveling, an agent carries process-specific code and data.

One simple example is the process of an expense report. An employee submits a list of expenses after a business trip. A mobile agent is generated with a list of tasks. After it collects all of the expenses reported, it generates a formatted expense report with the employee's name, list of expenses and total amount of money. It then travels to the group manager's computer for approval. After generating a simple message telling the manager its existence and how important it is, the agent suspends itself with all its states saved. When the manager has time to process the report, he reactivates the agent, get the report and approves it. The agent then accesses the project budget database and deducts the total amount of money on the report. The next stop is the finance department. The agent withdraws the total amount of money on the report and transfers it to the employee's account. Finally, it generates a receipt for the transaction.

### **5.3.4 An E-commerce Application**

Mobile agents are well suited for e-commerce. A commercial transaction may require real-time access to remote resources, such as stock quotes and agent-to-agent negotiation. Different agents have different goals and implement different strategies to accomplish them.

We will use our mobile agent system to implement the following e-commerce example. Kelly will go to Silicon Valley as a summer intern for three months. She needs to find a place to live during that time. An agent is created with her search criteria such as type of house,

facilities, location range, price range, etc. The agent travels to different advertisement sites and searches for those listings that meet her specification. After getting a list of items, the agent returns to Kelly and asks her to pick up one. After Kelly selects her favorite apartment, the agent moves to the site of the selected apartment's manager and asks more questions Kelly asked. If all answers are 'yes', the agent reserves the apartment and pays the deposit. It then goes back to Kelly and generates a report with detailed information about the apartment and a receipt of the deposit.

## 6 Specific Goals and Timeline

We will build a mobile code system with persistence support using the Voyager framework and PSE Pro persistent engine. This research will include the following works:

- **Class File Postprocessor:** The code transformation technique will be realized by class file postprocessing. The postprocessor will be implemented using JavaClass.
- **Formal Proof:** The correctness of code transformation for control state externalization will be formally proved using denotational semantics.
- **Examples and Demonstrations:** At least two examples will be implemented to demonstrate the effectiveness and correctness of the system and to make performance measurements. The first sample implementation is the DSS system described in Section 5. We will test multiple examples and measure their performances. After the programs are post-processed, they will be executed with and without shutdown. The second sample application is the expense report workflow system. The expense report agent will be implemented and post-processed. It will be executed on simulated distributed hosts. If time permits, a more complicated sample application may be implemented as well. It will be a e-commerce application using both servlets and mobile agents. The e-commerce example described in Section 5 is a possible choice.

Finally, the timeline for future work is shown in the table below:

May 1999	Proposal defense
July 1999	Finish implementing class file postprocessor
August 1999	Work out formal proof of code transformation
September 1999	Work out examples of applications
September 1999	Start drafting dissertation
December 1999	Dissertation draft complete
December 1999	Dissertation defense

## References

- [1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, April 1997.
- [2] M.P. Atkinson and R. Morrison. Orthogonally persistent object systems. In *VLDB Journal*, pages 319–401, 1995.
- [3] L. Cardelli. Obliq: A language with distributed scope. Technical report, Compaq Computer Corporation, System Research Center, May 1995.
- [4] L. Cardelli. Mobile computations. In *Mobile Object Systems: Towards the Programmable Internet*, pages 3–6. Springer-Verlag, April 1997.
- [5] W. Clinger and J. Rees. Revised report on the algorithm language Scheme. In *ACM Lisp Pointers 4*, pages 1–55, July 1991.
- [6] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *USENIX Annual Conference*, New Orleans, LA, June 1998.
- [7] M. Dahm. Byte code engineering with the JavaClass API. Technical report, Freie Universität Berlin, January 1999.
- [8] General Magic. *Odyssey Information*. <http://www.genmagic.com/technology/odyssey.html>.
- [9] M. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.
- [10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, September 1996.
- [11] D. Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge Computer Laboratory, June 1997.
- [12] Jon Howell. Straightforward Java persistence through checkpointing. In *The Third International Workshop on Persistence and Java(tm) (PJW3)*, September 1998.
- [13] IBM. *Aglets Specification 1.1 Draft*, September 1998. <http://www.trl.ibm.co.jp/aglets/spec11.html>.
- [14] M. Jordan and M. Atkinson. Orthogonal persistence for Java - a mid-term report. In *The Third International Workshop on Persistence and Java(tm) (PJW3)*, September 1998.

- [15] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report 93-05-06, Department of Computer Science and Engineering, University of Washington, May 1993.
- [16] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proceedings OOPSLA 98*, 1998.
- [17] T. Lindholm and F. Yellin. *The Java virtual Machine Specification*. Addison-Wesley, September 1996.
- [18] F. Matthes, S. Müssig, and J. Schmidt. Persistent polymorphic programming in Tycoon: An introduction. Technical report, Fachbereich Informatik Universität Hamburg, 1993.
- [19] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 403–414, Santiago, Chile, September 1994. VLDB Journal.
- [20] R. Morrison, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, D.S. Munro, and M.P. Atkinson. *The Napier88 Persistent Programming Language and Environment*. The FIDE Book, 1999.
- [21] Object Design Inc. *Bookshelf for ObjectStore PSE/PSE Pro Release 3.0 for Java*, 1998. <http://mothra.odi.com/content/products/pse/javadoc/index.html>.
- [22] ObjectSpace. *ObjectSpace Voyager Version 2.0.0 User Guide*, 1998. <http://www.objectspace.com/developers/voyager/white/index.html>.
- [23] P. O'Brien. Making Java object persistent. *Java Report*, pages 49–60, January 1997.
- [24] J. Plank and M. Puening. Checkpointing Java. Technical report, University of Tennessee at Knoxville Department of Computer Science, 1997.
- [25] Sun Microsystems. *Java Object Serialization Specification*, 1997. <http://chatsubo.javasoft.com/products/jdk/1.1/docs/guide/serialization/index.html>.
- [26] Sun Microsystems. *Servlet Specification, Version 2.1*, November 1998.
- [27] W. Tao and G. Lindstrom. Externalizing Java execution states. Submitted to OOPSLA 99, 1999.
- [28] W. Tao and G. Lindstrom. Externalizing Java execution states through exception handling. Submitted to IEEE Transaction on Software Engineering, 1999.
- [29] J. White. Telescript technology: Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.