# Sumatra: A Language for Resource-aware Mobile Programs [⋆]

Anurag Acharya, M. Ranganathan, Joel Saltz
Department of Computer Science
University of Maryland, College Park 20742
{acha,ranga,saltz}@cs.umd.edu

**Abstract.** Programs that use mobility as a mechanism to adapt to resource changes have three requirements that are not shared with other mobile programs. First, they need to monitor the level and quality of resources in their operating environment. Second, they need to be able to react to changes in resource availability. Third, they need to be able to control the way in which resources are used on their behalf (by libraries and other support code). In this chapter, we describe the design and implementation of Sumatra, an extension of Java that supports resource-aware mobile programs. We also describe the design and implementation of a distributed resource monitor that provides the information required by Sumatra programs.

## 1 Introduction

Mobile programs can move an active thread of control from one site to another during execution. This flexibility has many potential advantages. For example, a program that searches distributed data repositories can improve its performance by migrating to the repositories and performing the search on-site instead of fetching all the data to its current location. Similarly, an Internet video-conferencing application can minimize overall response time by positioning its server based on the location of its users. Applications running on mobile platforms can react to a drop in network bandwidth by moving network-intensive computations to a proxy host on the static network. The primary advantage of mobility in these scenarios is that it can be used as a tool to adapt to variations in the operating environment. Applications can use online information about their operating environment and knowledge of their own resource requirements to make judicious decisions about placement of computation and data.

Many systems provide some form of support for program mobility. The simplest form of support is the ability to download code and execute it to completion at a single site – as provided by systems like Omniware [3], Safe-TCL [6], Java [14]. Other systems like Avalon [11], NCL [13] REV [18] and Obliq [9] allow programs in execution to initiate computation on remote nodes and wait for their completion. The most sophisticated support is provided by systems like Agent

[⋆] This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

TCL [15], Emerald [17], Mole [24], Aglets [20], TACOMA [16] and Telescript [26] which permit an executing program to move while it is in execution.

Programs that use mobility as a mechanism to adapt to resource changes have three requirements that are not shared with other mobile programs. First, they need to be aware of their execution environment. In particular, they need to monitor the level and quality of resources in their operating environment. We refer to this as the *awareness* requirement. Second, they need to be able to react to changes in resource availability. We refer to this as the *agility* requirement. Third, they need to be able to control the way in which resources are used on their behalf (by libraries and other support code). We refer to this as the *authority* requirement.

In this chapter, we describe the design and implementation of Sumatra, an extension of Java that supports resource-aware mobile programs. We also describe the design and implementation of a distributed resource monitor that provides the information required by Sumatra programs.

We first discuss the constraints that the requirements of awareness, agility and authority place on languages for mobile programs. We then describe Sumatra and discuss how its design decisions have been guided by these constraints. Next, we discuss the considerations that guide the design of resource monitors, in particular, fault-tolerance and the need to allow applications to control the use of resources on their behalf. We describe Komodo, a distributed resource monitor and discuss how well its design meets the requirements. Both Sumatra and Komodo have been implemented and are available to individuals and organizations with a Sun JDK license. We provide implementation details wherever appropriate.

In this chapter, we assume that the reader is familiar with mobile code languages (like Java, Obliq, Agent-Tcl). Several chapters in this book provide excellent introductions to this class of languages and the mechanisms used to implement mobility.

## 2 Design constraints

In this section, we discuss the language design constraints that arise from the desire to be resource-aware and the use of mobility as a mechanism to adapt to changes in resource availability. We discuss the requirements of awareness, agility and authority and the constraints each of them generates. In the next section, we describe Sumatra and discuss how its design has been guided by these constraints.

### 2.1 Awareness

Resource-aware programs need to be able to monitor the availability and quality of the resources in their environment. A resource can be monitored either on-demand or continuously. Both kinds of monitoring are useful. On-demand monitoring is useful in three kinds of situations. First, if the resource in question is used infrequently but is expensive to use – e.g. an application on a mobile host that uses a cell-modem to periodically scan incoming mail being held at a post-office machine. Second, if the availability of the resource in question changes infrequently – e.g. an application that chooses the location from which it monitors a process based on the amount of disk space available at that location. Third, if the resource is expensive to monitor and the cost of monitoring outweighs the potential gains – e.g. an application that accesses large volumes of data over a very slow link. Continuous monitoring is useful if the resource is frequently used or if the resource changes frequently or is cheap to monitor.

On-demand monitoring is, by necessity, synchronous. The request for information does not return till the information is available. Continuous monitoring can, on the other hand, support both synchronous and asynchronous interfaces. A synchronous request would return immediately with information about the current availability of the resource in question whereas an asynchronous request would monitor the resource and notify the application only when the resource availability no longer satisfies an application-specified predicate. A synchronous interface is suitable, for example, for an application that sends a sequence of large messages and checks the state of the network before sending each message. An asynchronous interface is useful, for example, for an application that uses a resource continuously (like receiving a video stream over the network) or an application that does not use the network itself but calls library routines which may do so. An asynchronous interface is also useful to inform applications about a qualitative change in the operating environment, for example if the platform is mobile and may need to switch between multiple wireless networks [19].

For continuous monitoring, an important decision is the granularity at which resource change is to be monitored. The simplest alternative is to report every change to the requesting application. This is often impractical as most resource levels have some jitter which usually has little impact on application performance. The next simplest alternative is to use a jitter threshold and track only those changes that larger than this threshold. Jitter-threshold-based schemes have been proposed by several researchers as a way of dealing with resource variability on mobile platforms [7, 8]. Jitter-threshold-based schemes work well if changes in the resource levels are usually stable. Transient changes (usually just spikes) in the resource levels can cause spurious responses. The alternative is to augment the jitter-based scheme with a filter that eliminates transients. This allows the applications to track only the stable changes. It is therefore important to allow applications to register application-specific (or resource-specific) filters that determine which changes in resource availability/quality should be reported to the requesting application.

To summarize, languages for resource-aware mobile programs should provide a resource-monitoring interface that allows on-demand monitoring as well as continuous monitoring. For continuous monitoring, the resource-monitoring interface should allow programs to register a application-specific filter which determines which resource changes should be reported.

## 2.2 Agility

To achieve agility, a mobile code language should provide mechanisms that allow programs to react quickly to asynchronous events like revocation of allocated bandwidth/memory or qualitative changes in network connectivity (e.g. on a mobile host). Two mechanisms are required. First, the ability to receive the event in an asynchronous manner (a la Unix signals) and second, the ability to take appropriate action in response to such events including moving program execution to a different site.

The requirement that programs be able to move within event handling code constrains the choice of mobility mechanisms. There are two major alternatives: (1) a `go()` (or a `jump`) primitive that freezes execution at current site and resumes execution at target site; and (2) a function-call-like mechanism which allows programs to execute a procedure at a specified site. Most mobile code languages have selected one of these two alternatives. Agent-Tcl [15], Telescript [26] and Aglets [20] use a `go`-based mechanism whereas Obliq [9], Avalon [11], NCL [13] REV [18] and TACOMA [16] use a function-call-based interface. To ensure a prompt response to asynchronous events, a function-call-based interface would require one of two things: (1) either the language automatically captures the continuation at the point at which an event occurs and makes it available for use within the corresponding event handler; or (2) the programmer emulates this functionality by writing a large number of functions that represent continuations at different points in the programs. In both cases, the language has to allow the use of a continuation-passing style. We believe a `go()`-based interface is simpler to use; the astute reader has probably already noted that the first option has exactly the same effect as a `go()`.

## 2.3 Authority

The requirement of authority is, by far, the most demanding. It requires that language allow programs to control the way in which resources are used on their behalf by system support as well as by libraries. In effect, it requires that module boundaries not be completely opaque and that they allow resource-usage related restrictions to pass through and enforced as a part of execution.

There is a trade-off between the extent to which programs are allowed to control resource-use and ease of programming. At one extreme, a language can require that operations that use resources of interest be performed only after they have been explicitly authorized. This allows complete control over resource usage. Unauthorized accesses would raise an exception. A scheme similar to this is used by Java applets to control access to local resources (see Figure 1 for an example). This works well for Java applets as the level of access to local resources is not usually changed during the execution of a program and there is no advantage in retrying the operation.

On the other hand, it is entirely possible that the response to authorization failure for resource-usage can be (and in many cases, will be) to change the level of authorization. In such cases, retrying the operation would be desirable.

This can be a problem if authorization failures occur deep in library code. It may or may not be possible to restart the operation if the failure is delivered as an exception. For example, consider the code in Figure 2. For this example, assume that the resource of interest is the number of sockets. If the first call to checkCreateSocket() succeeds and the second fails (say the program has already created as many sockets as it was permitted to), an exception will be raised. A common response to this situation would be to negotiate with the resource manager for a higher limit on the number of sockets and to restart the operation. However, a clean restart for `createTwoSockets()` is not possible as the first socket has already been created.

This situation can be dealt with by treating resource-use violation as an asynchronous event and allowing programs to associate a handler with every restriction on resource-use. The handler would be executed in the same context as the operation that caused the violation and would allow the operation to be restarted, if that is what is desired.

To summarize, both forms of control over resource use are desirable: (1) checking all operations that might use resources of interest for authorization and delivering an exception if the usage is not authorized; (2) considering resource-use violation as an asynchronous event and allowing programs to associate event handlers to resource-use restrictions which execute in the same context as the faulting operation.

```
public boolean mkdir() {
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
    security.checkWrite(path);
  }
  return mkdir0();
}

public boolean renameTo(File dest) {
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
    security.checkWrite(path);
    security.checkWrite(dest.path);
  }
  return renameTo0(dest);
}
```

**Fig. 1.** Excerpts from Java class libraries that illustrate how access to local resources is controlled. The actual operations of creating a directory and renaming a file are performed by `mkdir0()` and `renameTo0()`.

```
public boolean createSocket(String host, int port, boolean stream) {
  ResourceManager resource = System.getResourceManager();
  if (resource != null) {
    resource.checkCreateSocket(host,port,stream);
  }
  return Socket(host,port,stream);
}

private Socket socket1;
private Socket socket2;
....
public void createTwoSockets(String host1,String host2,int port,
    boolean stream) {
...
  socket1 = createSocket(host1,port,stream);
  socket2 = createSocket(host2,port,stream); // fails authorization check
}
```

**Fig. 2.** Sample Java code to illustrate the problem with restarting operations that fail authorization checks. This code assumes that Java is extended with a Resource Manager similar to its current Security Manager.

## 3 Design and implementation of Sumatra

Sumatra is an extension of Java that supports resource-aware mobile programs. Platform-independence was the primary rationale for choosing Java as the base for our effort. In the design of Sumatra, we have not altered the Java language. Sumatra can run all legal Java programs without modification. All added functionality was provided by extending the Java class library and by modifying the Java interpreter, without affecting the virtual machine interface.

Sumatra adds four programming abstractions to Java: *object-groups*, *execution-engines*, *resource-monitoring* and *asynchronous events*. An object-group is a dynamically created group of objects. Objects can be dynamically added to or removed from object-groups. All objects within an object-group are treated as a unit for mobility-related operations. This allows the programmer to customize the granularity of movement and to amortize the cost of moving and tracking individual objects. Object-groups also allow the programmer to control the lifetime of objects. Objects that are included in an object-group continue to live on a host even after the thread that created them completes execution or migrates to some other host. Objects that do not belong to an object-group are subject to garbage-collection as usual.

An execution-engine corresponds to the notion of a "location" in a distributed environment. In concrete terms, it corresponds to an interpreter executing on a host. Multiple engines can exist on a single host. Sumatra allows object-groups to be moved between execution-engines. An execution-engine may also host active threads of control. Currently, multiple threads on the same engine are scheduled in a *run-to-completion* manner. We plan to implement other scheduling strategies in future. Threads can move between engines.

The resource-monitoring support in Sumatra allows programs to either query the level of resource availability or to control the extent to which various resources are used by the program itself as well as library code it is linked to. Both on-demand as well as continuous monitoring is supported. For continuous monitoring, the resource-monitoring interface allows programs to register jitter thresholds which determine which resource changes should be reported.

In Sumatra, asynchronous events are used to notify executing programs about urgent changes in their execution environment. These notifications can come either from the interpreter or from the external environment (the operating system or some other administrative process). We expect that the interpreter would use asynchronous events to notify the program about violations of resource-restrictions requested by the program itself; we expect that the external environment would use asynchronous events to inform the program about changes in the environment of the interpreter including resource revocation. Sumatra allows programs to register handlers for asynchronous events. The handlers for asynchronous events are able to inspect the current state of execution and can take appropriate action including moving away from the current execution site or changing the resource-restrictions in force.

In Sumatra, computation begins at a single site and spreads to other sites in three ways: (1) remote method instantiation, (2) remote thread creation, and (3) thread migration. Remote method instantiation corresponds to the familiar notion of RPC (remote-procedure-call) whereby the calling thread is suspended while an operation is performed, on its request, at a remote site. Remote thread creation differs from remote method instantiation in that the new thread is independent of the creating thread; the creating thread continues execution once the creation is complete. Finally, thread migration involves stopping the execution of the calling thread at the current site, transferring its state to another site and resuming execution at that site.

In the following subsections, we describe the design and implementation of Sumatra. The first three subsections describe the programming abstractions mentioned above. The final subsection discusses how the design decisions have been guided by the constraints described in the previous section.

### 3.1 Execution-engines

Execution-engines correspond to interpreters and are identified by a hostname and a port number that they listen on. They are created by specifying these parameters to the constructor. Several error conditions are possible – the remote host could be unreachable (due to a network partition), the remote host could be down, the remote host may not allow creation of interpreters, the desired port number could be in use. An exception is returned for each of these error conditions along with an error message that provides additional information.

Other possible error conditions include authorization errors. Execution-engines support three operations: thread migration, remote thread creation and downloading code. Figure 3 presents the execution-engine interface.

Sumatra allows explicit thread migration using a `go()` method that bundles up the stack and the program counter and moves the thread to the specified execution-engine. Execution is resumed at the first instruction after the call to `go`. To automatically marshal the stack, the Sumatra interpreter maintains a type stack parallel to the value stack, which keeps track of the types of all values on the stack. When a thread migrates, Sumatra transports with it all local objects that are referenced by the stack but do not belong to any object-group. Objects that belong to an object-group move only when that object-group is moved. Stack references to the objects that are left behind (i.e were part of some object-group) are converted to proxy references. After the thread is moved to the target site, it is possible that its stack contains proxy references that point to objects that used to be remote but are now local. These references are converted back to local references before the call to `go` returns. Several error conditions can occur during the execution of `go` – the remote host could be unreachable, the remote host may be down, the interpreter implementing the execution-engine might have died, the remote site may not have all the local classes that this program might need while executing on that site. An exception is returned for each of these error conditions along with an error message that provides additional information.

A new thread can be created by *rexec*'ing the `main` method of a class existing on a remote engine. The arguments for the new thread are copied and moved to the remote site. Remote thread creation is non-blocking and the calling thread resumes immediately after the `main` method call is sent to the remote engine. Note that remote calls to the `main` method are blocking – the calling thread is suspended till the execution of `main` completes. Remote thread creation is different from thread migration as it creates a new thread at the remote site that runs concurrently with the original thread; thread migration moves the current thread to the remote site without creating a new thread. Concurrent threads communicate using calls to shared objects. The thread creating a new thread can share objects with the child by passing it references to these objects as arguments to `main`.

Sumatra does not automatically move code for either the `go` operation or the `rexec` operation. The `downloadClass` method can be used to download the class template for an object (and the associated bytecode) to an execution-engine. This allows programs to control their environment to some extent – for each class, a program can decide whether to use its own implementation or an implementation provided by the host on which it is executing. Downloaded class-templates are cached; the `ClassLoader` checks this cache before checking the local file system. The host, however, retains complete control over which classes can be downloaded and can reject `downloadClass` operations that attempt to replace critical classes.

**Implementation** We use a mechanism similar to the familiar `inetd` daemon to manage the creation of execution-engines. A master daemon runs on all machines that allow creation of execution-engines and listens on a well-known socket. When a execution-engine creation request is received, it creates a new interpreter process which attempts to bind to specified socket. If either of these operations fail, an exception value is returned. The master daemon is also responsible for checking authorization. Currently, no authorization checks have been implemented. Execution-engine operations are currently implemented in C (as native methods). Figure 4 shows the interface for the Sumatra communication package which is used to implement these operations. Note that except `saveState()`, none of these *need* to be in C – `invokeRPC()` can be implemented using Java RMI [25] and the others can be layered on other Java classes. Java RMI was not used as it was not available when we were implementing Sumatra. Since Java does not provide a user-level primitive to serialize the stack, `saveState()` has to be implemented in C.

```
public final class Engine {
    public String hostname;
    public int port;

    public Engine(String hname,int portno);
    public void downloadClass(String Classname);
    public void go();
    public native void rexec(String classname, String[] args);
}
```

**Fig. 3.** The Sumatra engine interface

## 3.2 Object-groups

There are three primary properties of object-groups. First, they are aggregates. That is, they move only as a group. Second, they are sticky. That is, they can only be moved by an explicit move-object-group operation; they do not move with migrating threads. Third, they are persistent. That is, they are not garbage-collected. As long as an object is a member of some object-group, it is spared by the garbage-collector.

Figure 5 presents the Sumatra object-group interface. A string name can be associated, at creation time, with each object-group. The name can be used to identify different object-groups. Sumatra does not check for system-wide uniqueness of this name. It does, however, check for local uniqueness – attempts to create an object-group with a name that is in-use raises an exception.

```
public final class Comm {
    /* save state of current thread and transmit */
    public static native void saveState(String hostname,
            int portno);
    /* invoke an rpc call, walks stack to get arguments */
    public static native Object invokeRPC()
            throws ObjectMovedException;
    /* Download a class to an execution engine */
    public static native void downloadClass(String classname,
            Engine engine);
    /* Start an execution engine at a given machine */
    public static native void startEngine(String hostname,
            Engine engine);
    /* Find my current engine */
    public static native Engine myEngine();
}
```

**Fig. 4.** The Sumatra communication package

Objects can be dynamically added to or removed from an object-group using the `checkIn()` and `checkOut()` methods respectively. The `moveTo()` method is used to move the object-group to a different execution-engine. Membership of object-groups is explicit, that is, every member of an object-group must be checked in explicitly. Also, `moveTo()` does a shallow move – only the objects that have explicitly been checked in are moved. This is in accordance with the *authority* requirement – no communication takes place without an explicit request.

Thread objects cannot be checked into an object-group. This restriction is imposed as including a thread object in object-group would require the thread to move along with the object-group. This causes two problems. First, this could cause migration of executing threads at arbitrary points. Restricting migration to syntactically marked program points has advantages (see [2] for one such advantage). Second, since moving a thread could cause other objects to be moved, it would violate the *authority* requirement.

During a `moveTo()` operation, objects in an object-group are automatically marshaled using type-information stored in their class templates. When an object-group is moved, all local references to objects in the group (stack references and references from other objects) are converted into *proxy references* which record the new location of the object. Some objects, such as I/O objects, are tightly bound to local resources and cannot be moved. References to such objects are reset and must be reinitialized at the new site. Several error conditions are possible – the remote host could be unreachable (due to a network partition), the remote host could be down, the remote execution-engine may not contain the classes corresponding to the objects being moved and the remote execution-engine may already contain an object-group with the same name. An exception is returned for each of these error condition along with an error message that provides additional information.

Method invocations on proxy objects are translated into calls at the remote site. Type information stored in class-templates is used to achieve remote-procedure-call functionality without a stub compiler. Exceptions generated at the called site are forwarded to the caller. In accordance with the *authority* requirement, Sumatra does not automatically track mobile objects. Requesting a remote method invocation on an object that is no longer at the called site results in an *object-moved* exception at the calling site. To facilitate application-level tracking, the exception carries with it a forwarding address. The caller can handle the exception as it deems fit (e.g., re-issue the request to the new location, migrate to the new location, raise a further exception and so on). This mechanism allows applications to locate mobile objects lazily, paying the cost of tracking only if they need to. It also allows applications to abort tracking if need be and pursue an alternative course of action.

It is also possible for an application to specify that an exception should be delivered on all methods invoked on proxy objects. This allows the application to avoid communication if it so desires. Furthermore, Sumatra does not support direct access to instance variables of remote objects. Such variables should be accessed through remote invocation of access methods that return the values of instance variables. This is in keeping with the *authority* requirement.

**Implementation** Object-groups are implemented as lists of objects. Each execution-engine has a list of such lists. The object-group operations have been implemented in C (as native methods). The main reason for this is the interaction between object-groups and the garbage-collector. Objects that belong to an object-group are not garbage-collected. It is, however, possible to re-implement this feature portably using a keep-alive thread that lives forever and exists for the sole purpose of keeping these objects from being garbage-collected.

```
public final class ObjGroup {
   public String groupname;

   public ObjGroup (String name);
   public native void checkIn(Object object);
   public native void checkOut(Object object);
   public native void moveTo(Engine engine);
   public native Engine location();
   private native void internGroup();
}
```

**Fig. 5.** The Sumatra object-group interface

## 3.3 Resources and asynchronous events

Sumatra provides two resource-monitoring interfaces, one that allows the application to poll the state of a particular resource and the other that allows it to request asynchronous notification when the availability of a resource goes outside a specified threshold. Figure 6 presents the interface for on-demand monitoring. A program indicates its interest in a resource by creating a `Resource` object with appropriate arguments. If the resource to be monitored is associated with a single host (e.g. server load), the last argument is ignored. A request for continuous monitoring consists of a pair of values defining a range of resource availability (e.g. upper and lower bounds on bandwidth), an upper bound on frequency of polling and a function to be called when the resource availability is no longer within the range (this uses the interface for asynchronous events which will be presented shortly).

Sumatra provides a single interface (shown in Figure 7) for handling all kinds of asynchronous events. To create an event handler, the user creates a new subclass of the `Callback` class, creates an object of that class and registers it using `System.registerCallback()`. The event handler function is specified by overriding the `callback()` method. Note that this method has been declared to be `abstract` and every subclass has to override it. After the `callback()` method completes, control returns to the point where the interruption happened. If a call to `go()` is embedded in a `callback()` method, execution resumes on the target host.

Three kind of asynchronous events are currently supported – asynchronous notification for continuous resource monitoring, violation of resource-restrictions and external events in the form of Unix signals. The first class of events require upper and lower bounds and a frequency bound which are specified using the `setLow()`, `setHigh()` and `setFreq()` methods. The second class of events requires only an upper bound. The third class of events, Unix signals, require only the signal number. Signals can be used by the external environment (the operating system or some other administrative process) to inform the application about urgent asynchronous events, in particular resource revocation. Using a handler, the application can take appropriate action including moving away from the current execution site.

**Implementation** Sumatra assumes that a local resource monitor is available which can be queried for information about the environment. When an application makes a monitoring request, Sumatra forwards the request to the local resource monitor. If the monitor does not support the requested operation, or if no monitor is available, an exception is raised. The communication between Sumatra and the monitor is via a well-known shared memory segment. This allows Sumatra to cheaply acquire rapidly changing resource information. On-demand monitoring requests are implemented by directly reading this segment. For on-demand monitoring, default polling frequency of the resource monitor is used.

Event handlers need to be registered explicitly. Depending on the type of the event being registered, different structures are set up inside the interpreter. Asynchronous notification for continuous monitoring is implemented using Unix signals. The interpreter uses the same handler for all Unix signals - the identity of the signal is saved and an event is queued. The event queue is checked between every Java virtual machine instruction.

Resource-restrictions are implemented within the interpreter by activating counters that keep track of resource usage. These counters have been collected in a single module which is currently implemented in C. It is easy to re-implement this entirely within Java/Sumatra, much along the same lines as the Security Manager. Currently, only the "memory-use" restriction is implemented.

```
public final class Resource {
   public Resource(String type, String from, String to);
   public int read_value();
}
```

**Fig. 6.** The Sumatra on-demand monitoring interface.

```
public abstract class Callback {
   /* these callbacks are used for asynchronous notification */
   public Callback(Resource rsid);
   /* these callbacks are used for external event handlers */
   public Callback(int type);
   public void SetLow(int low);
   public void SetHigh(int high);
   public void SetFreq(int freq);
   public Resource get_resource();
   abstract public void callback();
}
```

**Fig. 7.** The Sumatra event handling interface.

### 3.4 Discussion

In this section, we discuss the ways in which the design of Sumatra is influenced by the *agility* and *authority* requirements.

- **Calls to `go()` can occur anywhere:** In particular, they can be embedded inside callback methods. This allows Sumatra programs to react quickly to asynchronous events like revocation of allocated bandwidth/memory or qualitative changes in network connectivity (e.g. on a mobile host). The design alternative to using a `go()`-like interface was to allow migration only at function-call boundaries. To ensure a prompt response to asynchronous events, this would require one of two things: (1) either the language automatically captures the continuation at the point at which an event occurs and makes it available for use within the corresponding callback method ; or (2) the programmer emulates this functionality by writing a large number of functions that represent continuations at different points in the programs. In both cases, the language has to allow the use of a continuation-passing style. We believe, the `go()`-based interface provided by Sumatra is both simpler to use and easier to implement.
- **All remote accesses can be trapped:** there are two parts to this. First, Sumatra does not allow programs to access instance variables of remote objects. Attempts to do result in an exception being raised. Second, programs can request that an exception be raised for all methods invoked on proxy objects. Using both these features, a program can turn off all communication if it so desires.
- **Objects moved and tracked in groups:** this allows application to control the granularity of both operations.
- **Object-groups are tracked lazily:** and under application-control. Requesting a remote method invocation on an object that is no longer at the called site results in an *object-moved* exception at the calling site. The exception carries with it a forwarding address which allows the application to continue tracking by re-issuing the request to the new location or to abort tracking if it so desires.
- **Membership of object-groups is explicit:** only those objects that have been explicitly added to an object-group belong to it. If an object with one or more component objects is added to an object-group, only the top-level object becomes a member of the object-group. To include the component objects in the object-group, each of them has to be explicitly added. This allows the application to precisely control which objects are moved and when.
- **No distributed garbage-collection:** Sumatra provides no distributed garbage-collection. It is the responsibility of the application to ensure that objects that are no longer needed are removed from object-groups. Note that Sumatra does provide local garbage-collection.
- **Object-groups are sticky:** Objects that belong to an object-group move only when that object-group is moved. When a thread migrates, Sumatra transports with it all local objects that are referenced by the stack but do not belong to any object-group.
- **Life-time of an object can be controlled:** Object-groups and their member objects are not subject to garbage-collection. This is implemented by adding the list of object groups to the set of roots used by the garbage-

collector. This ensures that all objects that belong to object-groups (and their transitive closure) are spared by the garbage-collector. This allows applications to temporarily deposit data at intermediate execution sites as well as to extend existing servers by downloading objects that extend current functionality. It is legal for an object to belong to multiple object-groups. Membership of multiple object-groups can be useful in situations where there are multiple reasons to keep an object alive. This can, however, lead to unexpected communication if one of the object-groups moves and takes the object with it. It is, however, the programmer's responsibility to ensure that multiple membership does not lead to unwanted communication.

- **Location of an object can be queried:** Sumatra allows programs to query for the location of an object. This allows programs to selectively control communication – if desired, a program can allow critical remote operations while restricting all other remote operations.
- **Memory use can be bounded:** Sumatra programs can specify an upper bound on memory allocation. Attempts to allocate memory beyond this bound results in an asynchronous event which be handled if so desired.

### 3.5 Example

In this section, we provide a feel for the Sumatra programming model using a simple example. The task is to scan through a database of X-ray images stored at a remote site for images that show lung cancer. This task can be performed in two steps. In the first step, a computationally cheap pruning algorithm is used to quickly identify lungs that might have cancer. A compute-intensive cancer-detection algorithm is then used to identify images that actually show cancer.

One way to write a program for this task would be to download all lung images from the image server and do all the processing locally. If the absence of cancer in most lung images can be cheaply established, this scheme wastes network resources as it moves all lung images to the destination site. Another approach would be to send the selection procedure to the site of the image database and to send only the "interesting" images back to the main program. If the selection procedure is able to filter out most of the images, this approach would significantly reduce network requirements. A third, and even more flexible, approach would allow the shipped selection procedure to extract all the interesting images from the database but return only the *size* of the extracted images to the main program. This information can be be used, in conjunction with information about network bandwidth between the current location and the database site to estimate the transfer time for the selected images. If the estimated time is too large, the program may choose to move itself to the database site and perform the cancer-detection computation there rather than downloading all the data. This avoids downloading most images at the cost of (possibly) slower processing at the server. On the other hand if the transfer time is small enough, the data can be shipped over and processed locally. Figure 8 shows code for the third approach.

```
.....
filter_object = new Lung_filter();
cancer_object = new Lung_checker(filter_object);
myengine = System.comm.myEngine();

// Create a engine at the xray database site.
remote_engine = new Engine("xrays.gov");
// Indicate interest in monitoring bw to xrays.gov
bw = new Resource("bandwidth",myengine.hostname,"xrays.gov");
// Send the lung_filter class to the remote engine
remote_engine.downloadClass("Lung_filter");
// Create a new object group.
objgroup = new ObjGroup("lung_filter_group");
// Add the lung_filter_object to the object group
objgroup.checkIn(filter_object);
// Move the object group to the database site
objgroup.moveTo(remote_engine);

// a remote method call selects interesting xrays
size = filter_object.query(db,"DarkLungs");

// compute estimated time to transfer images
transfer_time = size * bw.read_value();
// Does it take too long?
if (transfer_time > threshold) {
  // Migrate thread, process images and return.
  remote_engine.go();
  result = cancer_object.detect_cancer();
  myengine.go();
}
else {
  // the estimated transfer time is small enough
  // Fetch them and process locally.
  objgroup.moveTo(myengine);
  result = cancer_object.detect_cancer();
}

// display result locally
System.display(result);
```

**Fig. 8.** Excerpt of a Sumatra program that migrates depending on the time required to transfer data.

Sumatra assumes that a local resource monitor is available which can be queried for information about the environment. In the next section, we describe Komodo, a distributed resource monitor which can provide information for Sumatra applications.

## 4 Komodo: a distributed resource monitor

For different applications, different resource constraints are likely to govern the decision to migrate - for example network latency, network bandwidth, server load (as in number of server connections available), CPU cycles etc. We have propose that a single monitor be used for all resources. Using a single monitor facilitates applications that might need information about multiple resources. It also reduces communication requirements for distributed monitoring as information about multiple resources can be sent in the same message.

In our design, each host runs a monitor daemon which communicates with peers on other hosts. The monitoring daemons are loosely-coupled and use UDP for communication as well as for monitoring the network. A simple timeout-based scheme is used to handle lost packets and re-transmissions.

Applications register monitoring requests with the local daemon. If the resource mentioned in the request can be monitored from the current host then the local daemon handles the request. Requests that cannot be handled locally, for example, network latency between two remote sites, are forwarded by the local daemon to the daemon on the appropriate host.

Applications can request the current availability of a resource (on-demand monitoring) or they can request periodic checks on resource availability (continuous monitoring). On-demand monitoring returns a single snapshot. Continuous monitoring applies a resource-specific filter to eliminate jitter in resource levels. Eliminating jitter helps reduce the reporting requirements (and therefore the communication needed) without impacting application performance. Data corresponding to remote requests for continuous monitoring is forwarded to the requesting sites as and when the filtered value of the resource changes. Requests for continuous monitoring may also specify a sampling frequency, subject to an upper bound. Komodo enforces an upper bound on this frequency to keep the monitoring cost at an acceptable level.

Each daemon supports a limited number of monitoring requests. This limit applies to both local and remote requests. Together with the limit on sampling frequency, this ensures the monitoring load on individual hosts is within acceptable limits.

Each request has an application-specified *time-to-live*. There is an upper bound on the *time-to-live* which allows the daemons to clean-up requests made by applications or hosts that have since crashed. Applications need to refresh requests within the *time-to-live*. Requests that are not refreshed are dropped. If a daemon runs out of entries in its monitoring table, the least recently requested entry is ejected.

Monitoring requests are passed from Sumatra to the local Komodo daemon using a well-known Unix domain socket. The resource information is made available by the daemon in a read-only shared memory segment. This allows applications to rapidly access the latest available monitoring information.

## 4.1 Implementation

The current implementation of *Komodo* monitors network latency. Each Komodo daemon pings a network link for which it has received monitoring requests, by sending a 32-byte UDP packet to the daemon on the other end of the link of interest. If an echo is not received within an expected interval, (the maximum of the ping period or five times the current round trip time estimate) the packet is retransmitted.

To eliminate short-term variation in latency measures, we developed a filter based of an extensive study of Internet latency [1]. This study revealed that: (1) there is a lot of short-term jitter in the latency measures but in most cases, the jitter is small; (2) there are occasional jumps in latency that appear only for a single observation; and (3) In most cases, a short window of values around the mode contains a large fraction of the observations (this indicates that the mode would be a good characteristic value for RTT distributions). Based on this, we have developed a filter for latency measures that returns a moving window mode if there is a well-defined mode, else it returns a moving window mean.

We plan to extend Komodo, in the near future, to monitor network bandwidth and server load (number of available server connections).

## 4.2 Discussion of the design of Komodo

In this section, we discuss the design of Komodo and how it has been influenced by the *awareness* and the *authority* requirements.

Komodo provides both on-demand and continuous monitoring. It allows applications to select the monitoring mode on a per-resource and a per-host basis (or host-pair basis for network latency and bandwidth). For resources that are monitored in a continuous mode, Komodo allows applications to control the frequency with which the resource is monitored. This allows the applications to control how much effort is spent in monitoring on their behalf and is in keeping with the *authority* requirement. To safeguard hosts from malicious or runaway applications, Komodo enforces an upper bound on the monitoring frequency.

Komodo provides both a synchronous interface and an asynchronous interface. In the current implementation, asynchronous notification is implemented using UNIX signals. For the asynchronous interface, Komodo allows resource-specific filters which pre-process the information about individual resources; individual applications do not have to replicate this functionality. They can, however, control its operation by providing their own values for the filters' parameters. This allows the applications to track only the stable changes.

For network-related resources, there is the choice of active versus passive monitoring. Komodo provides only active monitoring. It assumes that passive monitoring is the responsibility of the applications. For applications that do not coordinate their operations, this can lead to some loss of information. For example, if two applications independently access the same server, they could share information about the network connection to server but if they have not been designed to cooperate in this manner, they will not be able to utilize the information acquired by the other. Since Komodo is a user-level monitoring system, it can keep track of network performance for other programs in only two ways: (1) applications are required to measure performance for every network access and supply the information to the local Komodo daemon or (2) all messages are routed through Komodo. Neither of these are attractive options. We believe that cooperating applications should share such information instead of requiring all applications to acquire/provide it. Such an approach was been used by Mummert et al [22] in the *Coda* file system which is able to adapt to changes in network connectivity. Individual components of *Coda* cooperate in monitoring the bandwidth and maintain the information in a shared location.

## 5 Discussion

Currently, much of the core support for resource-awareness has been implemented in C. As has been discussed in individual sections, much of this can be re-implemented as Java/Sumatra libraries with relative ease. In particular, the resource-restrictions can be implemented using a Resource Manager similar to the current Security Manager module. It is, however, not possible to re-implement all of Sumatra in a portable manner. The primary limitation is the lack of a portable way to save and restore the stack. Another limitation is support for handling external events like Unix signals.

Process migration and remote execution have been proposed, and have been successfully used, as mechanisms for adapting to changes in host availability [10, 12, 21, 27]. Remote execution has also been proposed for efficient execution of computation that requires multiple remote accesses [11, 13, 18] and for efficient execution of graphical user interfaces which need to interact closely with the client [5]. Both these application scenarios use remote execution as a way to avoid using the network. Most proposed uses of Java [14] also use remote execution to avoid repeated client-server interaction. In these applications, decisions about the placement of computation are hard-coded. To the best of our knowledge, Sumatra (together with Komodo) is the first system that allows distributed applications to *monitor* the network state and *dynamically* place computation and data in response to changes in the network state.

Network-awareness is particularly important to applications running on mobile platforms which can see rapid changes in network quality. Various forms of network-awareness have been proposed for such applications. Application-transparent or system-level adaptation to variations in network bandwidth has been successfully used by the designers of the Coda file system [22] to improve the performance of applications. The Odyssey project on mobile information access plans to provide support for application-specific resource monitoring and adaptation. The primary adaptation mechanism under consideration is change in data fidelity [23]. Athan and Duchamp [4] propose the use of remote execution for reducing the communication between a mobile machine and the static network. In all these systems, location of the various computation modules is fixed; adaptation is achieved by changing the way in which the network is used.

Two other recent Java-based systems, Aglets [20] and Mole [24] provide some support for mobility. Both these systems have been implemented using Java and Java RMI and as such are unable to provide true thread migration. They provide only object mobility. A weak form of process migration can be achieved by programmers by explicitly unwinding the stack and copying whatever is needed into appropriate instance variables. Both systems provide an explicit restart method that is called to start the processing at the destination site. Sumatra provides true thread migration but requires C code to do so. This distinction can be eliminated if future versions of Java provide a portable way to pack the stack. Other major features of Sumatra that are not provided by Mole and Aglets are object-groups, support for application-level tracking of object-groups (or objects in their case), control over lifetime of objects (which is needed to implement distributed garbage-collection at the application-level). Neither system provides resource-monitoring facilities. Nor do they provide the capability to cleanly move within exception handlers (it is hard to unwind the stack in such a situation). Finally, Aglets proposes asynchronous mobility requests. Sumatra guarantees that threads will not be moved without an explicit request.

A point to note is that the constraints introduced by the *awareness* and *authority* requirement are common to all resource-aware programs, in particular programs that execute on resource-limited platforms like mobile computers. Noble, Price and Satyanarayanan [8] as well as Badrinath and Welling [7] propose notification-based schemes for tracking resource changes for mobile hosts. Both propose an interface that allows applications to specify a jitter threshold. The constraints introduced by the *agility* requirement, however, are specific to mobile programs.

## Acknowledgments

We would like to thank Shamik Sharma for many thought-provoking discussions. We would like to thank Mustafa Uysal for being our resident skeptic.

## References

1. A. Acharya and J. Saltz. A Study of Internet Round-Trip Delay. Technical Report CS-TR-3736, University of Maryland, December 1996.
2. A. Acharya and J. Saltz. *Dynamic Linking for Mobile Programs*, chapter unknown. Springer Verlag, 1997. Jan Vitek and Christian Tschudin (eds).
3. A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 127–36, May 1996.

4. A. Athan and D. Duchamp. Agent-mediated Message Passing for Constrained Environments. In *Proceedings of the USENIX Mobile and Location-independent Computing Symposium*, pages 103–7, Aug 1993.
5. K. Bharat and L. Cardelli. Migratory Applications. In *Proceedings of the Eighth ACM Symposium on User Interface Software and Technology*, pages 133–42, Nov 1995.
6. N. Borenstein. Email With a Mind of its Own: The Safe-TCL Language for Enabled Mail. In *Proceedings of IFIP Working Group 6.5 International Conference*, pages 389–402, Jun 1994.
7. B.R.Badrinath and Girish Welling. Event Delivery Abstractions for Mobile Computing. Technical Report LCSR-TR-242, Rutgers University, 1996.
8. Brian D. Noble and Morgan Price and M.Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing*, Feb. 1995.
9. L. Cardelli. A Language With Distributed Scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, Jan. 1995.
10. J. Casas, D. Clark, R. Konuru, S. Otto, and R. Prouty. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.
11. S. Clamen, L. Leibengood, S. Nettles, and J. Wing. Reliable Distributed Computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, pages 169–79, 1990.
12. F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, Aug 1991.
13. J. Falcone. A Programmable Interface Language for Heterogeneous Systems. *ACM Transactions on Computer Systems*, 5(4):330–51, Nov. 1987.
14. J. Gosling and H. McGilton. The Java Language Environment White Paper, 1995.
15. R. Gray. Agent TCL: A Flexible and Secure Mobile-agent System. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, July 1996.
16. D. Johansen, R. van Renesse, and F. Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical Report 95-23, University of Tromso, 1995.
17. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(2):109–33, Feb. 1988.
18. J.W. Stamos and D.K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.
19. R. Katz. The Case for Wireless Overlay Networks. Invited talk at the ACM Federated Computer Science Research Conferences, Philadelphia, 1996.
20. D. Lange and M. Oshima. *Programming Mobile Agents in Java.* In progress, 1996. (ch 2,3).
21. M. Litzkow and M. Livny. Experiences with the Condor Distributed Batch System. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, Al., 1990.
22. L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 1995.
23. M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware Adaptation for Mobile Computing. *Operating Systems Review*, 29(1):52–5, Jan 1995.

24. M. Straβer, J. Baumann, and F. Hohl. Mole - A Java Based Mobile Agent System. In *Proceedings of the ECOOP'96 workshop on Mobile Object Systems*, 1996.
25. Sun Microsystems. *Java Remote Method Invocation. http://chatsubo.javasoft.com/current/rmi/index.html.*
26. J. White. Telescript Technology: Mobile Agents, 1996. *http://www.genmagic.com-/Telescript/Whitepapers.*
27. E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 13–24, Nov. 1987.

This article was processed using the LaTeX macro package with LLNCS style