TOS: A Kernel of a Distributed Systems Management System

Kåre J. Lauvset* Dag Johansen* Keith Marzullo†

March 2, 2000

Abstract

Distributed systems are becoming harder to manage, in part because the uses we put to distributed systems are rapidly changing. Hence, the software used to manage a distributed system needs to be flexible enough to accommodate these new uses. It also has to be secure enough to not allow unauthorized changes to be made to the system.

We present a library and a kernel that supports the management of distributed systems. This kernel, called the TACOMA Operating System (TOS), is an outgrowth of our research into mobile agent systems and applications. We have used TOS to implement several novel distributed management policies, one of which is discussed in detail in this paper. We conclude the paper with a discussion on the applicability of mobile agents to self-managing distributed applications.

1 Introduction

Distributed systems are ubiquitous. The operations of virtually all organizations ranging from small companies to international corporations depend on networks of computers. Distributed computer systems are rapidly becoming a critical corporate resource.

Most of these organizations are facing the problem of managing their distributed systems. It is rarely possible to set up a collection of machines with a collection of software and leave this set unchanged for a long period of time. For example, even if a company attempts to continue to use one version of a document system, employees may receive external documents composed using a later version of the same document system. To be able to read these external documents, a company may find itself forced to move forward to the newer version. Another difficulty arises from heterogeneity. High-performance computers are becoming ever more inexpensive, and the cost of increasing the size of a distributed system is low. Such increases, however, exact a cost in maintenance, since newer machines run software that often cannot be run on older machines. A third difficulty arises with software upgrades. For example, distributed services such as authentication servers, clock synchronization, and web cache servers can be hard to upgrade without first bringing the entire distributed system down.

The difficulty of distributed systems management (which we abbreviate as *DiSM*) have led to a number of projects (for example [2, 8]) and products (for example Zero Administration for Windows by Microsoft, Zero Effort Networks by Novell, NetWizard Plus by Attachmate and Patrol by BMC Software). Most have concentrated on one key problem: the distribution, installation, and upgrading of software in a heterogeneous distributed system. This problem, while key, is not the only hard problem that needs addressing. For example:

- Some programs, like many network services, are distributed. Changing a distributed program can require synchronized shutdown, installation, and start-up. Given the criticality of many distributed system, the outage time during this change over should be made as small as possible.

- The legal ramifications of having pirated code are severe and the laws are becoming more stringently enforced. A system administrator

*Department of Computer Science, University of Tromsø, Tromsø, Norway. This work was supported by NSF (Norway) grant No. 112578/431 (DITS program).

†Department of Computer Science and Engineering, University of California San Diego, La Jolla 92093-0114, California, USA. In doing this work, Marzullo was supported by NSF (Norway) grant No. 112578/431 (DITS program).

may find it necessary for legal reasons to periodically sweep the system for unlicensed software.

- E-commerce applications are growing in use. The load being generated by e-commerce clients is similarly growing and attacks targeted to e-commerce sites are becoming frequent. Specialized software to aid in the diagnosis and repair of the system (for example, determining emerging system resource bottlenecks or detecting novel intrusion attempts both from within or outside of the company) may need to be installed, upgraded, or removed.

The above list is, of course, not exhaustive. Indeed, because the uses of distributed systems are rapidly changing, it is hard to identify what the key DiSM problems will be a few years from now. The DiSM system should be designed to be flexible enough to accommodate existing and future management policies.

Rather than developing a suite of new DiSM services as an end in itself, we have built a library, called the *TACOMA Operating System*, or *TOS*, that allows for the rapid deployment of new and flexible DiSM services and the secure modification of existing services. We have designed the runtime support of TOS to be small enough to fit on any programmable internet device, and we have developed several DiSM policies using the TOS library.

This paper proceeds as follows. In Section 2 we describe requirements of a DiSM system. Section 3 describes the TOS library and runtime, and Section 4 illustrates the use of our system by giving a detailed example of a DiSM policy we have implemented. In Section 5 we discuss our system in the context of distributed systems management before we present the conclusions in Section 6.

2 Requirements of a DiSM System

One normally thinks of DiSM system requirements in terms of the function that it provides: installing software, monitoring load, and so on. Such a function is easily implemented using TOS. We believe, however, that there are three non-functional requirements that DiSM systems should meet:

1. There is a bootstrapping problem with deploying a DiSM system: the DiSM system cannot initially install itself. The initial core of the system should therefore be small, portable across different platforms, and easily installed using traditional techniques.

2. The DiSM system should not open new security holes. If care is not taken, then remote software installation can be used to install malevolent software as well as it can install required software.

3. The DiSM system should be able to manage itself. It should accommodate novel and complex management policies, and accommodate specific requirements and restrictions of different system administrators. Operationally, it should be able to add and remove its own functional modules.

There are several different software systems that meet some of these requirements. For example, the Java Virtual Machine is a good platform upon which to address the first requirement. A mobile agent system can satisfy the third requirement. Many DiSM problems can be solved efficiently with specialized protocols that implement the required synchronization among the machines being managed. Encoding the protocol as a mobile agent makes the deployment of these specialized protocols easy [23]. Thus, it is not surprising that there has been some effort into applying mobile agents to network management [5]. The second requirement, however, has not yet been satisfactorily solved by mobile agent systems.

The original impetus for TOS was to make a mobile agent system that is suitable for real mobile agent applications including DiSM. We have gone through several experimental cycles in the design and application of TOS. We have found that the above requirements have changed the way we think of mobile agent systems. The first version (called TNT) was a port of the original TACOMA to Windows NT and the Win32 API. In the second version (also called TNT) we separated the `meet` primitive into several primitives. One reason for doing so was to separate the mechanism for mobility from inter-agent communication. Further, the format of a marshalled briefcase was changed and was binary compatible with the TACOMA version for UNIX. The third major version, called TOS, was implemented using Java and JVM and is the direct predecessor of the current version. The most important changes that were introduced with TOS were related to the structure of both mobile agents and the programming model. For example, in the current version, we have introduced the term *carrier* instead of mobile agents to clarify the intended usage. In addition, the core API is collected into a separate library. Currently, the TOS library consists of 9 Java classes about 12Kb of Java byte code.

One observation we have made is that distributed systems management does not need all of the functions that are associated with a full-blown mobile agent system. Hence, TOS supplies only the minimal amount of function needed to support its extension to the functions that are actually required. In this sense, we have borrowed from the work on extensibility in operating system kernels (e.g., [18, 4, 6, 21]) by including in TOS only the mechanisms needed to securely implement different mobility functions. Doing so helps TOS meet the third requirement listed above.

For example, we have found it rare for a DiSM policy to require an itinerant agent—that is, an agent that can move from machine to machine following a trajectory it determines on the fly [10]. Many recent systems that allow for self-managed distributed systems (e.g. Elastic Servers [8], Extensible Servers [10] and Active Servers [1]) supply what would be considered "single-hop" mobility. And, reasoning about the effects of an itinerant agent can be hard. Hence, TOS mobile agents are not by default itinerant. However, if a system administrator wishes to implement a DiSM policy using an itinerant agent, then the required support can be included with TOS.

Another observation we have made is that DiSM policies are often most efficiently implemented when the mobile agent primitives are used sparingly. A simple example is software download: a mobile agent is a convenient mechanism for sending a rule to a machine that determines whether or not a software package should be downloaded, but *ftp* or *scp/ssh* are good mechanisms for copying files. In general, a DiSM mobile agent is best split into one part that implements the required synchronization and local testing, and another part that consists of transfer, installation, and other configuration management functions. We call the second part the *carrier* of the DiSM policy. Carriers are written in Java and reference the TOS library. The remaining part of a DiSM policy can be written in any language, and can often be supplied by existing software tools on the target machine.

3 TOS

In this section we describe the TOS architecture. We first define some terminology, and then describe the major components of a TOS kernel. We conclude by describing the TOS library and give an example of a simple TOS enabled application.

3.1 Terminology

A *carrier* is a specific kind of mobile agent: it is written in Java and derives from the `Carrier` class of the TOS library. Carriers are used to carry programs (written in any language) and data to remote machines. Carriers also implement the logic and the synchronization needed to implement a DiSM policy. They are much like the Xerox worm [20] which was separated into two functional units: a control structure that determined which machine to target and a payload that it started at the new machine.

A *TOS kernel* is a running Java virtual machine that accepts carriers. It provides the abstraction that one associates with *places* [24], *firewalls* [13] and *landing pads* [11]. It differs from places and landing pads in that it is extensible. Specifically, a TOS kernel can include *extensions*, each of which provides additional functions available to carriers (and to other extensions) executing at that kernel.

A TOS kernel need not be dynamically extensible, but a TOS kernel can have an extension that allows other extensions to be downloaded and removed. Such an extension would provide the desired authentication and authorization required by the system administration policies. For example, one can implement an extension that downloads other extensions that originate from a system administrator and that are digitally signed by the corporate security officer.

Readers familiar with the original TACOMA [14] systems should be aware that the structure of TOS is different. In designing TOS, we have concentrated on distilling the minimal set of functions

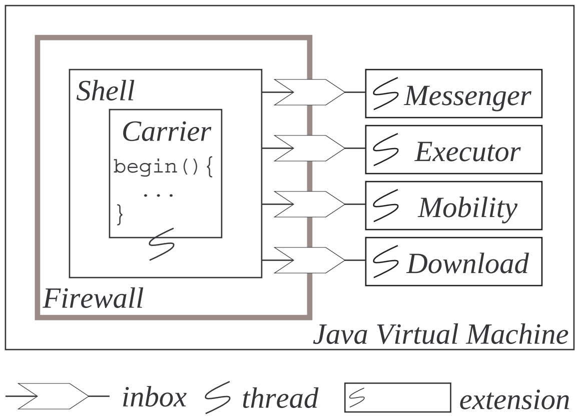

Figure 1: TOS kernel structure.

needed for a mobile agent system. In doing so, we have separated the mobility aspects of a mobile agent (which is the carrier) from the functional aspects. The TACOMA `briefcase` abstraction has also been separated into three different TOS abstractions. This is mainly because TOS has been designed and implemented using an object-oriented approach. The `CODE` folder, which is the main thread of control of a TACOMA mobile agent, has become the Java code that the carrier executes. The `HOST` folder, which specifies the itinerary of the TACOMA mobile agent, has become the TOS `Path` class. A Path differs from the `HOST` folder in that TOS assigns no meaning to its contents. Instead, it is interpreted by an extension. The extension can exert control, such as not allowing a carrier to move to a TOS kernel outside of some administrative domain. The remaining folders in a TACOMA briefcase are instead held by an instance of the TOS `Data` class.

3.2 TOS Architecture

Like a carrier, a TOS kernel is also written as a Java program that references the TOS library. Figure 1 is a schematic illustration of a TOS kernel. We explain the use of the various components of this illustration below.

3.2.1 Extensions and the Firewall

Extensions are threaded objects that are included with a TOS kernel to extend what a carrier can do. For example, Figure 1 shows a kernel with four extensions: one that implements a service that sends messages to other instances of itself at remote TOS kernels, one that executes programs outside the kernel, one that opens connections to a well-known location and downloads files, and one that accepts serialized versions of carriers and starts running them in the kernel.

TOS leverages off the security mechanisms provided by the Java Virtual Machine (JVM). A JVM can have a *security manager* that is invoked whenever a Java thread attempts to execute a potentially dangerous operation, such as reading or writing a local file or creating a new thread (the former can destroy user data, and the latter can lead to a denial of service attack in the JVM). TOS provides a security manager called the *firewall* that prohibits carriers from executing any potentially dangerous operation. For carriers, the firewall is equivalent to the default JVM security manager, which excludes all dangerous operations. The firewall, however, allows extensions to execute any operation. The distinction between carriers and extensions is made using Java thread groups: the default group is used for extensions and another group is used for carriers. If required, more fine-grained restriction on executing dangerous operations can be easily done by, for example, extending the firewall to use more thread groups to represent different classes of services.

There is a limit, however, in how much can be gained by leveraging off the JVM security mechanisms. [3, 15] For example, a thread can theoretically allocate any amount of free memory. Doing so might lead to a denial of service for another thread in the same JVM. And, a thread cannot be forced to terminate without its cooperation, so once a resource is allocated it cannot, in the general case, be preempted. In general, denial of service attacks can not be prevented in Java/JVM. By using Java, TOS inherits these weaknesses. Such attacks, of course, only affect the behavior of carriers and not the programs that they carry.

The security of a TOS kernel depends upon the extensions that it runs. One could easily design an extension that misused its abilities to execute dangerous operations. Hence, we imagine that the most secure way to use TOS is to have a few generic extensions rather than many specialized extensions. If this is done, then each extension can be carefully designed and checked for possible security weaknesses. And, with fewer extensions, it is easier to examine them for ways that one might interfere with another.

Examples of extensions we have built include:

Atomic commit. The *Commit* extension allows a set of carriers to participate in either a one-phase or a two-phase atomic commit protocol.

Extender. This extension installs or removes extensions from the TOS kernel.

Executor. This extension runs a program in a new process on behalf of a carrier. The extension returns a handle for the process that is running the program.

File downloading. The *Download* extension retrieves files named by URLs.

Inter-kernel communications. The *Messenger* extension allows communication between extensions and carriers that are running in different TOS kernels.

Mobility extensions. We have implemented two: single-hop mobility (which provides a remote evaluation facility [22]) and itinerant style mobility (which is the most frequently used mobility pattern of mobile agent systems). Mobility extensions should use some form of authentication; ours only accept carriers that have a valid digital signature to ensure that they come from a trusted source.

Norwegian Army Protocol. This extension (NAP) provides a primary-backup like failure detection and recovery scheme for carriers [12].

Spawner. This extension creates a copy of the carrier that invokes this extension and moves that copy to the processor that is the first element of the path.

3.2.2 Folders and Inboxes

A carrier can, if desired, directly invoke an extension's public methods. In fact, a carrier can directly invoke a public method of any object for which it can obtain a reference. We have found, though, that using direct method invocation can often be inconvenient. For example, a carrier rarely needs to block waiting for the action an extension takes on its behalf to complete. Multiple carriers can request service from an extension concurrently. This requires synchronization, for example through synchronized and public methods. To allow the carrier to continue as quickly as possible, the extension's public method should enqueue the requests thereby holding the critical section lock for as short a period as possible. Another problem has to do with protection: an extension may wish to restrict some communication to occur only with other extensions. Hence, TOS provides another mechanism for communications, called *inboxes*, that avoids direct method invocation. Rather than being based on invocation, it more closely resembles message passing. While a method invocation mechanism might be used that meets these requirements, message passing is more convenient [16].

As in TACOMA, a TOS *folder* is a ⟨key, value⟩ pair. Folders are used to hold and carry carrier-interpreted data. A carrier can create any number of folders. An inbox is a uniquely named queue of folders. A carrier can enqueue one folder at a time on an inbox, and an extension can dequeue folders from any inbox that it created. When creating an inbox, an extension can specify that only other extensions are allowed to enqueue folders on that inbox. This restriction allows extensions to be able to communicate without interference from carriers. This illustrates another advantage of inbox-based communication.

All of the extensions that we have implemented use inbox-based communications. Figure 1 shows four inboxes, one for each extensions with which carriers communicate.

3.2.3 Mobility and the Shell

A carrier is derived from the TOS `Carrier` class. This class contains one abstract method, called `begin`. A carrier begins executing at a TOS kernel by having its `begin` method invoked. When this method returns, the appropriate mobility extension takes control and transfer the carrier to the next entry in its path if such a move is legal (and if the extension supports itinerant style mobility).

The representation of a carrier in transit is defined by a TOS class called `Shell`. The shell is a runtime context in which a carrier executes, and it provides a method that writes the carrier to a Java `OutputStream`. The constructor for Shell requires a reference to a `Carrier` object (used when creating a new carrier). The static method `load` loads a carrier from a Java `InputStream` (used, for example, when receiving a carrier from another Java program). The shell captures any exception that is raised by the carrier it wraps, and so serves as a simple form of fault containment.

A carrier does not immediately start executing once the shell that wraps the carrier is created. Before starting its execution, its properties—namely its path, its data, and its byte codes—can be inspected and modified by an extension having a reference to the shell. For example, an extension could insert a security automaton into the carrier [7] or modify the carrier's path for the purpose of scheduling[1] [11]. The extension starts the execution of the carrier by an explicit method call to its shell: no code of the carrier, including its constructor, is executed before this method is invoked.

Assuming the default mobility service, the flow of control of moving a carrier is as follows. The carrier terminates when it returns from its `begin` method. This method returns no values, but it may throw an exception. The mobility extension waits for the shell to signal that the carrier has completed. Assuming no exception is thrown, the mobility extension extracts the agent's path and removes the first element. It then creates a TCP/IP connection to the named machine using a well-known port and creates a Java `OutputStream` on top of this connection. The mobility service then uses the shell to save the carrier to the `OutputStream`.

On the receiving machine is a mobility extension that, when a TCP/IP connection request arrives, creates a Java `InputStream` and hands it to a new thread of the extension. This new thread creates a new shell using the `InputStream` and starts the carrier.

[1]At the least, this extension should check the carrier's name to ensure that it does not start with "tacoma." or "java.", since if it did the carrier could be loaded as part of a package critical to the security of TOS.

3.3 TOS Library

In this section we describe the important classes of the TOS API. We first show how to write a carrier, then how to launch a carrier, then how to write an extension, and finally how to assemble a TOS kernel.

3.3.1 Writing a Carrier

Figure 2 shows a simple *Postman* carrier that posts a folder into the inbox of an extension. The folder contains a message that is displayed on the machine at which the TOS kernel is running.

This example does not illustrate the power of using carriers. A more realistic example would have the carrier determine which version of a program to download based on the configuration of the machine on which it is running. Such a carrier would be more complex since it needs to determine the configuration, but its structure would be essentially the same as the one in figure 2.

```
(1)  public class Hello extends Carrier {
(2)    public void begin() {
(3)      new Folder(null, "Hello, World").post("Echo");
(4)    }
(5)  }
```

Figure 2: A simple *Postman* carrier.

A carrier is written by extending the `Carrier` class (line 1). It must supply the `begin` method, which serves as the *main* function of the carrier. The carrier creates an unnamed folder (line 3; the name is the first parameter to the constructor) that has as a value the string "Hello, World". The carrier then posts this folder to the inbox named "Echo" (line 3). Note that when inbox-based communication is used, the carrier does not name an extension, but rather its associated inbox.

When `begin` returns, the carrier moves to the next TOS kernel listed in its path. In this simple example, the carrier does not modify its path.

3.3.2 Launching a Carrier

Figure 3 shows a fragment of a Java program that launches the carrier described above. The carrier is sent to the TOS kernels at *host0*, *host1* and *host2*.

To launch a carrier, a program first creates an instance of the carrier's class and also creates a `Shell`

```
(1)  ...
(2)  Shell sh = new Shell(new Hello());
(3)  String[] pa = { "host0", "host1", "host2" };
(4)  sh.path.set(pa);
(5)  Socket so = new Socket(sh.path.next(), 21212);
(6)  OutputStream os = so.getOutputStream();
(7)  sh.save(os);
(8)  os.close();
(9)  ...
```

Figure 3: Launching a carrier into the network.

object (line 2) that wraps the carrier. The program then sets the carrier's path (line 4). The launching program does not use this shell to run the carrier (although it could), but rather to send a serialized version of the carrier (line 7) to the TOS kernel running on the first machine in the carrier's path (line 5).

3.3.3 Writing an Extension

Figure 4 shows the code of the extension that created the "Echo" inbox. It waits for folders to be placed into this inbox and then prints their contents to standard output.

```
(1)  public class Echo implements Runnable {
(2)    public void run() {
(3)      Inbox ib = new Inbox("Echo");
(4)      while (true) {
(5)        while (ib.size() > 0)
(6)          System.out.println((String)ib.next().data);
(7)        try {
(8)          synchronized(ib) { ib.wait(5000); }
(9)        } catch (Exception e) {
(10)         e.printStackTrace();
(11)       }
(12)     }
(13)   }
(14) }
```

Figure 4: A simple display extension.

An extension is written by implementing the `Runnable` interface (line 1), or by extending the Java class `Thread`. An extension implements the `run` method (line 2), which is the entry point for the thread associated with the extension. Inboxes are created by class instantiation (line 3), and folders are retrieved via the `next` method on inboxes (line 6). The number of folders in an inbox can be determined using the `size` method (line 5). Posting a folder to an inbox causes the inbox to be notified, and so the extension waits for that notification (line 8).

3.3.4 Assembling a TOS Kernel

Figure 5 shows how to build a TOS kernel that supports the execution of the carrier of Figure 2.

```
(1)  public class Kernel {
(2)    public static void main(String[] args) {
(3)      new Thread(new Echo()).start();
(4)      new Thread(new Mobility()).start();
(5)    }
(6)  }
```

Figure 5: A simple TOS kernel.

A TOS kernel is relatively easy to build. One defines an object (line 1) that, when started (line 2) creates a thread for each extension included with the kernel (lines 3 and 4). Each TOS kernel should include some mobility extension (line 4) since without it no carrier can move to this kernel.

4 A DiSM Example

An important part of distributed systems management is intrusion detection, and so we give an example of using TOS from this field. One kind of attack is called *doorknob rattling*. This type of attack exploits a net-accessible vulnerability. For example, the *Ping of Death* attack exploits a vulnerability to receiving echo request datagrams with a data area of more than 65,507 octets [9]. Systems that have this vulnerability may crash upon the receipt of such an oversized echo request datagram.

Not all administrators are prompt in closing such vulnerabilities, and so a viable attack method is to write a script that sweeps across a set of machines trying to exploit the vulnerability. Even if an administrator has closed the vulnerability, it is still useful to know that such an attack is being attempted: the administrator can use this knowledge to try to trace back the source of the attack and to get some idea of how exposed his machines are.

To detect a doorknob rattling attack, there needs to be a *sensor* program that can detect the attempt to exploit a vulnerability on the machine on which it is running, and a *correlator* program that collects information from the sensors. The administrator would probably wish to run the correlator on a trusted machine. The correlator is, most likely, fairly simple and general—it monitors for a sequence of reports from sensors running on ma-

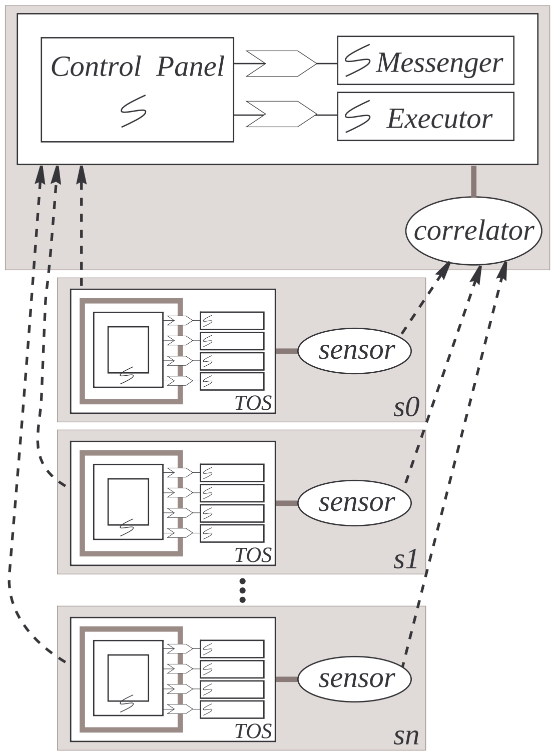

Figure 6: Distributed doorknob rattling monitor.

chines listed in a local configuration file—while the sensor is most likely specific—it is modified by the administrator to test for an attempt to exploit a specific vulnerability.

The role of TOS is to act as the glue that configures the distributed application, monitors its execution, and causes (when requested) its controlled termination. Figure 6 shows the application in the deployed state. In this figure, machines are shown as shaded rectangles, processes are shown as white rectangles or ovals. The white rectangles represent Java virtual machines running TOS kernels. A solid line from a white rectangle to an oval indicates that the rectangular process created the oval process. Communication is shown as dashed arrows.

The TOS components of this application are:

Install carrier. A carrier is used to install and monitor the execution of a sensor. Let S be the set of machines to be monitored. A carrier is launched to each machine s in S. Once arriving at s, the carrier copies over the sensor using the *Download* extension and starts it executing using the *Executor* extension. The carrier then reports that the sensor has started via the *Messenger* extension. At this point, the carrier periodically reports via the *Messenger* extension that the sensor is still running. The carrier learns through the *Messenger* extension when it is time to stop and deinstall the sensor program, after which the carrier terminates.

Security is maintained via the *Download* extension; only programs from a well-known and controlled location are downloadable, and the transmission of the program is authenticated and protected via *ssh*. Similarly, the *Executor* extension only executes programs that have been downloaded via *Download*.

The above solution is satisfactory for small S. If S is large, then this solution may put an unacceptably large load on the trusted processor due to the large number of simultaneous downloads. If this is the case, then S can be divided into k roughly equal subsets $S_1, S_2, \ldots, S_k$ for some value of k. Then, only k carriers are launched. Carrier i is launched to the first machine in S_i. This carrier copies the sensor from the trusted processor and then uses the *Spawner* extension to create a copy on the second machine in S_i. This process continues until a copy of the carrier is running on each machine in S_i.

Control Panel. The Control Panel is a Java application that allows the system administrator to control the deployment of the sensors, to monitor their current status, and to terminate their execution. The Control Panel is built using TOS but, like the launcher described in Section 3.3.2, is neither a carrier nor an extension. It uses the *Messenger* extension to receive information about the status of the sensors, and to notify the carriers which sensors should terminate execution.

While not necessary, it is convenient to have the Control Panel also be the application that launches the carriers and that starts the correlator running via the *Executor* extension. By doing so, the administrator needs to run only a single application to deploy and start monitoring the system.

Having the configuration management separated from the function of this application (that is, monitoring for doorknob rattling attacks) allows one to easily turn the above solution into a proactive monitoring approach. The Control Panel can be rewritten to capture the standard output of the correlator when it starts it executing via the *Executor* extension. By doing so, the Control Panel can actively alert the system administrator when an attack is detected and can initiate some suitable response. For example, the administrator can be notified using an SMS (Short Message System) service to his cellular telephone, and the IP address A of the machine that is originating the attack can be sent to another program running on each processor in S. This program would then exclude all communications originating from A (for example, by adding A to the *host.deny* file).

5 Discussion

We have presented TOS as a kernel for DiSM systems. It provides a mechanism for writing DiSM policies. Furthermore, it allows the DiSM system to be modified as the demands on the system change. TOS is configured using a modular structure so that extensions can be easily added, removed, or replaced at runtime without compromising the security of the system. In this way, TOS shares the same extensibility benefits as microkernels and extensible kernels.

Distributed systems management, however, need not be done only by a system administrator. A distributed application can be self-managing. Such applications are becoming more attractive as the scale of and the load on distributed applications are rapidly growing.

The processes of a distributed application use some protocol that has been defined specifically for that application. For example, the doorknob rattling detection application of Section 4 uses a simple protocol through which the sensors report information to the correlator. This specialized protocol is deployed by installing the sensors and the correlator, by writing a configuration file for the correlator to read, and by running these programs.

What TOS provides is an easy way to build and deploy another application specific protocol, namely the protocol that controls the deployment, redeployment, and monitoring of the application itself. Having this second protocol be separate from the application itself makes good software engineering sense, because the functional aspects of the application are usually independent from the configuration aspects. Using TOS gives one the same ease of use and flexibility that was provided by module configuration languages such as C/Mesa [17] that were used for building non-distributed applications. Distributed applications run in environments that are more complex than those for non-distributed applications, and the languages that have been developed for configuration management in distributed systems are correspondingly more complex (for example [19]). Rather than crafting a specialized (and more complex) language, we use a general-purpose language that can encode protocols specific to the application at hand.

We believe that this separation of concerns—functional from non-functional aspects in distributed applications—is an ideal application for mobile agents. By being mobile, mobile agents can easily deploy the non-functional protocol, and by using a full-fledged language, the mobile agents can implement the configuration and monitoring protocol. Using TOS encourages the programmer to separate these two concerns—functional from configuration and monitoring—and it allows the systems administrators to restrict how applications can be run in their particular administrative domains.

6 Conclusion

We have presented TOS, which is the kernel of a distributed systems management system. TOS includes a Java library and a set of extensions that can be used to implement a mobile agent system. We have described its architecture, given a simple example in detail, and shown how a more complex DiSM policy can be built. We also have argued that TOS can be used to easily build self-managing distributed applications, which we believe is an ideal application of mobile agents.

TOS is available for downloading from `http://www.tacoma.cs.uit.no/`.

Acknowledgements We would like to thank the other members of the TACOMA team. The ideas

behind TOS are based on the efforts of the whole team.

References

[1] Elan Amir, Steve McCanne, and Randy Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *ACM SIGCOMM Conference*, Vancouver, British Columbia, Sep 1998.

[2] Yair Amir, David Breitgand, Gregory V. Chockler, and Danny Dolev. Group Communication as an Infrastructure for Distributed System Management. In *The International Workshop on Services in Distributed and Networked Environment*, pages 84–91, Jun 1996.

[3] Godmar Back and Wilson Hsieh. Drawing the Red Line in Java. In *The 7th IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, Mar 1999.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, Dec 1995.

[5] Andrzej Bieszczad, Bernard Pagurek, and Tony White. Mobile Agents for Network Management. *IEEE Communciations Surveys*, 1(1), Sep 1998.

[6] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, Colorado, Dec 1995.

[7] Ulfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. Technical Report TR99-1758, Department of Computer Science, Cornell University, Jul 1999.

[8] G. Goldszmidt and Y. Yemini. Distributed Management by Delegation. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 333–341, Jun 1995.

[9] B. Harris and R. Hunt. TCP/IP Security Threats and Attack Methods. *Computer Communciations*, 22(10):885–897, Jun 1999.

[10] Dag Johansen. Mobile Agent Applicability. In *Proceedings of the Mobile Agents*, pages 80–98, Stuttgart, Germany, Sep 1998.

[11] Dag Johansen, Keith Marzullo, and Kåre J. Lauvset. An Approach towards an Agent Computing Environment. In *The Workshop on Middleware at the International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[12] Dag Johansen, Keith Marzullo, Fred B. Schneider, Kjetil Jacobsen, and Dmitrii Zagorodnov. NAP: Practical Fault-Tolerance for Itinerant Computations. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[13] Dag Johansen, Nils P. Sudmann, and Robbert van Renesse. Performance Issues in TACOMA. In *Proceedings of the 3rd Workshop on Mobile Object Systems, 11th Europeean Conference on Object-Oriented Programming*, Jyväskylä, Finland, Jun 1997.

[14] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical Report 95-23, Department of Computer Science, University of Tromsø, 1995.

[15] Danny B. Lange and Mitsuru Oshima. Mobile Agents with Java: The Aglet API. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration - An edited Collection*. Addison Wesley, 1999.

[16] H.C. Lauer and R.M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, 1979.

[17] Hugh C. Lauer and Edwin H. Satterthwaite. The Impact of Mesa on System Design. In

Proceedings of the Fourth International Conference on Software Engineering, 1979.

[18] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, Copper Mountain, Colorado, Dec 1995.

[19] O.G. Loques, J.C.B. Leite, A. Sztajnberg, and M. Lobosco. Towards Integrating Meta-Level Programming and Configuration Programming. Technical Report CAA no. 1, Instituto de Computaao, Universidade Federal Fluminense, Feb 1999.

[20] John F. Shoch and Jon A. Hupp. The "Worm" Programs - Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, Mar 1982.

[21] Christopher Small and Margo Seltzer. Structuring the Kernel as a Toolkit of Extensible, Reusable Components. In *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems*, pages 134–137, Lund, Sweden, Aug 1995.

[22] James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, Oct 1990.

[23] Jim Waldo. The Jini Architecture for network-centric computing. *Communications of the ACM*, 42(7), Jul 1999.

[24] Jim White. Mobile Agents White Paper, 1996.