Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-003

Robustness and Security in a Mobile-Agent based Network Monitoring System

Anand Tripathi, Muralidhar Koka, Sandeep Karanth, Ivan Osipkov,
Harsha Talkad, and Tanvir Ahmed, David Johnson, Scott Dier

January 13, 2004

Robustness and Security in a Mobile-Agent based Network Monitoring System *

Anand Tripathi, Muralidhar Koka, Sandeep Karanth, Ivan Osipkov, Harsha Talkad,
Tanvir Ahmed, David Johnson, and Scott Dier
Department of Computer Science
University of Minnesota, Minneapolis MN 55455

Abstract

In Konark, a network monitoring system based on mobile-agents, agents can communicate with each other to perform system-wide correlation of data. To minimize management efforts, our system incorporates mechanisms to detect and self-recover from internal failures in a decentralized and scalable fashion. In this paper, we discuss the mechanisms for self-recovery achieved by using the same mechanisms as those used for monitoring computing resources in the network. Self-monitoring of Konark also provides all the features of network monitoring, such as dynamic extensibility, active monitoring, and online-correlation of data. The security mechanisms of Konark are also discussed. This work demonstrates that mobile-agent based approach is a viable alternative for building robust and secure network monitoring systems.

Keywords: Self-Monitoring and Recovery, Network monitoring, Mobile agents, Multi-agent systems, Mobile code, Monitoring system security, Distributed Event Communication

1 Introduction

Monitoring of computing systems in an organization is required to ensure proper functioning of system services by detecting inconsistencies in system configuration and to protect system resources from being misused by users or attacked by intruders. Monitoring and managing large networks in an organization is becoming increasingly complex due to several factors. A large number of resources need to be monitored; frequent addition of new hardware and software components requires inclusion of new monitoring capabilities; moreover, vast amount of monitored data is collected from diverse sources, which needs to be processed and correlated in a timely manner.

* This work was supported by National Science Foundation grant ANI 0087514.

We present here the robustness and security mechanisms in Konark [12], a mobile agent based system for monitoring computing infrastructure in an organization. Because a monitoring system is an important facility to alert the system administrators in a timely manner of any abnormal conditions, its robustness is an important requirement. For a large-scale environment, managing and configuring the monitoring system itself should require very little administrative efforts. For this reason, the monitoring system itself needs to be resilient and it should be *self-healing* to perform on its own the recovery of its failed components. In our design, the primary emphasis is on the repair and recovery of failed components to ensure continuous operation of the monitoring system.

For ease of management, the monitoring system should be self-configurable. When a new resource is installed in the monitored environment, the system should automatically start monitoring it according to the system-wide policies. The monitoring system should also be secure to prevent attackers from disabling or bypassing it. Finally, the resource consumption overheads imposed by the monitoring system should not hinder the functioning and performance of users in the monitored environment.

Our work illustrates that a mobile-agent based approach is a natural solution to implement monitoring systems [12]. It provides a convenient mechanism to transport code to a host to perform a desired set of monitoring functions. A mobile-agent represents an object capable of migrating in a network to perform certain designated tasks at one or more nodes [4]. In this approach, mobile-agents are sent to continuously monitor the nodes in a network, detect and generate events, perform event correlations locally, and notify other system components of any significant events. As mobile-agents are *first-class objects*, their state and behavior can be altered remotely by invoking methods on them. These features can be used to make the mobile-agents dynamically extensible and securely modifiable. At a high level, our monitoring system consists of agent servers that run on monitored hosts and execute mobile-agents launched from administration servers. The agents encapsulate the

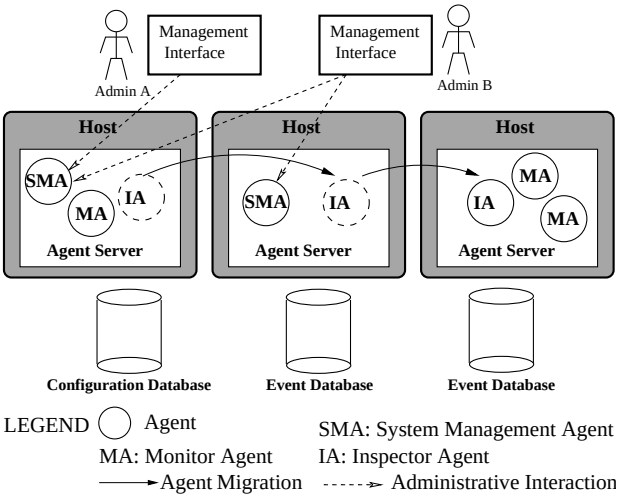

Figure 1. Monitoring system organization

logic necessary to detect various events. The architecture and basic building blocks are presented in [12].

The primary focus of this paper is on the self-monitoring and self-protection capabilities of our monitoring system. An important aspect of our design is the use of the monitoring system's inherent capabilities to detect its own component failures. The same set of mechanisms are used for both monitoring the computing infrastructure resources as well as the components of the monitoring system itself.

In Section 2 we present the architecture and the distributed event detection/aggregation model of our monitoring system. In Section 3 we present self-monitoring mechanisms to detect and recover from the failures of the monitoring system's components. We outline the various modes of failures, and requirements for recovery and repair of failed components. In our approach, system-wide self-monitoring is built upon component-level self-monitoring for scalable operations of the monitoring system. Section 4 presents the security mechanism for protecting the monitoring system's components. These mechanisms can enforce the desired security policies, which may be altered dynamically based on the occurrence of certain events in the system. Section 5 briefly outlines the capabilities of the configuration that we have installed and experimented with in our lab environment.

2 Overview of the Network Monitoring System

The Konark network monitoring system is implemented using Ajanta [6], a secure Java-based framework for distributed programming with mobile agents. There are three main components in Ajanta framework: *agent*, *agent server*, and *name registry*. Ajanta facilitates creation of

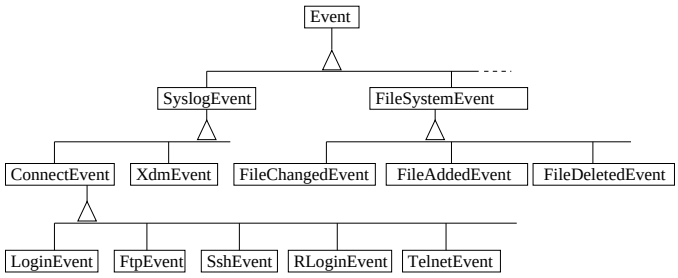

Figure 2. Event hierarchy

agents with a set of programming primitives to control agent migration patterns and protection of resources carried by an agent. An *agent server* hosts agents and provides them restricted access to local resources based on their credentials. All globally accessible entities are given location-independent names maintained by the name registry.

2.1 Monitoring System Components

The Konark monitoring system is implemented based on the *publish-subscribe* paradigm [12]. Every host in the monitored environment runs an agent server to support the installation and migration of agents. Primary functionality of the agents in the monitoring system is event detection, which is done through data collection, filtering, and correlation. A typical monitoring configuration is shown in Figure 1. Agents in the monitoring systems play different roles based on their functions.

System Management Agent (SMA) are primarily responsible for managing system-wide monitoring policy based on *monitoring configurations*. An SMA typically runs on secure hosts, launches Monitor Agent (MA) to hosts based on monitoring configurations, and remotely controls and modifies these agents. The SMA can migrate to a different host and resume its management functionalities. It can also be replicated for fault tolerance. It checkpoints the monitoring configuration, whenever it is updated by the administrators.

A monitoring configuration specifies the hosts and system components that need to be monitored, agents that should monitor these components, detection procedures for each agents, events generated as part of the detection procedures, and event subscription policies for event-correlation. *Configuration database* maintains and checkpoints system-wide monitoring configurations. System administrators interact with SMAs through *management interfaces*. Specific types of events, *alarm events*, are pushed to management interfaces.

2.2 Event Model

A *basic event* represents a significant change in the state of an entity being monitored. Detection of such events can be done locally on a host. Higher level *compound events* are derived from these basic events by correlating them. These events are given different alert levels, based on their severity. We use a canonical definition and representation of events, independent of operating system specific details. The event hierarchy is presented in Figure 2. The *SyslogEvent* class represents events generated from the system log files. Its subclass, *ConnectEvent* class represents various kinds of connections to a host.

2.3 Monitor Agent Architecture

The most important functions of an agent fall into three broad categories: execution of local detector functions, sending of event notifications, and subscription of events generated by other agents. The architecture of a monitor agent is shown in Figure 3. A monitor agent is extended from Ajanta's base *Agent* class and implements the *RemoteControlInterface* to ensure that the agent can be remotely controlled for operations, such as *recall*, *retract*, or *terminate*. It implements the *Monitor* interface to facilitate remote modification of monitoring policies. Event detectors, handlers, and subscribers corresponding to an event are registered through the monitor interface. Functional capabilities of a monitor agent are controlled remotely by SMAs. The *Subscriber* interface is implemented by an agent that subscribes to events from other agents.

A monitor agent is launched with a variety of *detectors* for detecting basic events at its host, as specified in the monitoring configuration. Event detection could include parsing log files or monitoring system resources. A detector can be triggered by events that are generated by other detectors. Such triggering dependencies are maintained in a *trigger table*. Associated with each detector is a handler object, which sends the generated events to their subscribers. If specified in monitoring configuration, a handler object could store the events in local or remote databases.

Each detector contains a thread, which executes the detection procedure asynchronously with other activities in the agent. All events generated by the detectors as well as events received from remote agents are deposited in the *event queue* to be processed by the agent's *event processing thread*. In step 1 of Figure 3, the event processing thread waits for an event to arrive in the event queue. On the arrival of an event, it first executes the event specific handler, and then through the trigger table initiates asynchronous execution of dependent detectors. These are the steps 2, 3, and 4 in Figure 3. The event handlers send the events to all currently registered subscribers. Monitor agents can subscribe

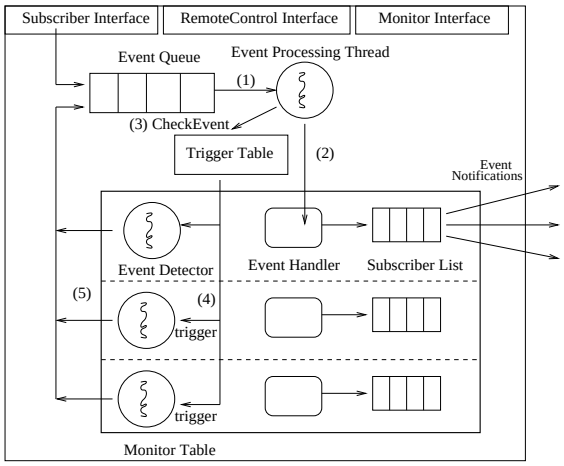

Figure 3. Architecture of a monitor agent

or resume subscriptions to events from other agents and cancel or suspend existing subscriptions for other agents.

2.4 Migratory Inspector Agents

An inspector agent travels to different hosts based on its itinerary, performing periodic configuration and consistency checks. It generates events and notifies other agents, as required by the monitoring configuration policy. In our monitoring system, File Integrity Checker (FIC) is a type of inspector agent that checks if system files or other important files have been tampered with. The File Integrity Checker visits hosts in the network, records file attributes and file content checksum. This information is sent as an event to a monitoring agent that has a detector for checking file integrity. During the initialization round, the detector stores the events in a database. In subsequent rounds, any inconsistencies in the received events generate events such as, FileDeleted, FileModified, which in turn may trigger alarms for the administrator. In order to remain lightweight, the inspector agent sends data to the monitoring agent from each host before migrating to the next. In an alternative approach with a single-hop model, the SMA launches an inspector agent to each node in the monitoring system. These inspector agents reside at the monitored nodes and send data to the monitoring agent periodically. In contrast to the previous case, the SMA needs to keep track of all the launched inspector agents whether they are alive or not. In the multi-hop scenario the SMA needs to keep track of just a single agent.

2.5 Example of Operational Capabilities

In our current implementation, monitoring agents process log files at each hosts, monitor system files integrity,

and fingerprint host configuration including typical services and routing table. Network-wide events are correlated to find attacks, e.g., user's switch to multiple accounts, and login from black-listed domains. At each host, agents monitor processes running with root privileges, runaway processes, i.e., zombie processes or those consuming resources over predefined thresholds, failure of system level services such as termination of daemons, and execution of malicious programs such as packet sniffers, and password crackers.

3 Robustness of the Monitoring System

Our goal is to reduce burden on system administrators in configuring, managing, and maintaining monitoring systems so that their attention is focused more on higher level policies and critical abnormal conditions. Our monitoring system has many components whose failure and recovery need to be addressed. In a typical installation, each host runs one agent server, and each such server has at least one agent performing local monitoring functions. An agent may also store certain events on various database servers for persistent storage and subsequent correlations with later events. The monitoring agent at a host typically contains over 20 different kinds of detectors for monitoring various aspects of a host such as log files, user processes, disk systems, and daemons processes. New detectors can be installed on an agent. An agent server may also periodically receive itinerant mobile agents performing periodic configuration checks.

Our design was driven with the goal of performing automatic recovery and repair of failed components within a short time-span, typically in the range of a few minutes. This duration needs to be kept short so that an attacker cannot complete an attack without getting noticed. The primary objective is to perform recovery and repair of failed components to ensure continued monitoring operations rather than diagnose failures. However, any repeated failure-recovery cycles during short intervals may suggest an attack on the monitoring system itself. Such situations can be detected using the existing mechanisms to perform correlations of frequent failure conditions.

Our monitoring system achieves robustness by incorporating mechanisms for *self-monitoring* and *self-configuration* at different levels of the system architecture. The event detection, correlation, and notification mechanisms presented in the previous section are used as the basic building blocks for failure detection. The publish-subscribe model is used for notifying failure conditions to other network components that need to participate in recovery and repair. Our design uses the notion of continuous periodic detection and notification of a failure event until the failed components causing it are repaired.

3.1 Failure Modes of System Components

The failure modes in our system are host failure, agent server failure, agent failure, and detector failure within an agent. Agent servers or hosts may crash bringing down the entire set of monitoring mechanisms on that node. The monitoring system requires at each host certain system level service daemons to be running. The SSH daemon at a host is critical for the System Management Agent and recovery agents to remotely start or shut down the agent server at that host. The RMI registry process must always be running for agents to receive RMI requests. Therefore, any failures of these two service daemons should be detected in a timely manner. Recovery from certain failures, such as the SSH daemon crash, require human intervention; on the other hand, agent server and RMI registry can be restarted automatically by the remote agents performing recovery actions.

A host crash implies crash of its agent server and all agents running on it. Agent servers may also crash independent of their host crashes, or they may fail partially. For example, the agent transfer protocol may fail, but the agents currently residing could continue to function. A crash of an agent server implies loss of all agents and their monitoring functions at that host. An agent server crash is detected during the recovery of its failed agents or by an agent trying to migrate to that server. The recovery of an agent server requires restoring all agents that are permanently installed at that server for performing monitoring functions. An itinerant mobile agent lost due to the server crash is recovered by other mechanisms.

The agents themselves could fail in many different ways, and in many cases the causes of the failures would be unpredictable. An agent may completely fail and stop communicating with other components. An agent may also incur partial failures. Each detector, which runs as a separate thread within an agent, could fail and stop performing its monitoring functions. In our system, a failed detector can be remotely un-installed and replaced with a new one by a remote agent performing recovery actions.

An agent's RMI interface may stop functioning either due to failure within the agent or due to the termination of the RMI registry daemon process. An agent may also notice failures of a remote agent when trying to communicate with it through its RMI interface. On such failures, as part of RMI exception handling mechanism, it retries the operation after once again looking up the RMI registry entry for the target agent. This is to recover from situations when an agent was restarted and has created a new RMI interface for itself. In the same way, a failure of the Ajanta name registry is detected through its RMI interface.

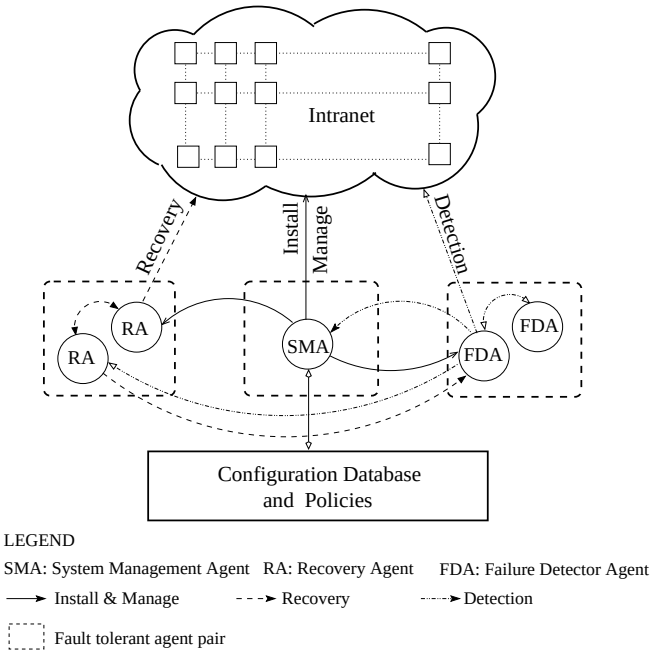

Figure 4. Recovery architecture

3.2 Recovery Requirements and Design Issues

Our design has been driven with the goal of minimizing the time for which a resource remains un-monitored, and keeping it close to 1 or 2 minutes. The state of an agent consists of various detectors and its publisher-subscriber relationships with other agents in the system. Some part of this state is checkpointed by SMAs, whereas the remaining part is maintained as *soft state* and reconstructed through interactions with other agents during recovery.

An agent's checkpointed configuration consists of the list of detectors that it should contain and the events generated by other agents that it should subscribe to. Information about any new subscriptions registered by an agent during its execution is not checkpointed. This forms the soft state, which is recreated on restart, as detailed below. The motivation for this is to keep the checkpointing overhead low.

Most of the detectors are either state-less or they maintain soft state, i.e. they are designed to reconstruct the state when restarted. The failure of an agent does not mean that all events monitored by that agent are lost. Many of the events are typically logged locally by the host operating system in various log files. Such events are retrieved by the agent when it is restarted. For example, the soft state of a detector processing log file events consists of the position offset in the log file for the last processed event. On restart, the detector determines the most recent event that it has processed. It obtains this information by querying the event databases.

A crash of an agent may result in abortion of a detector's execution that may be processing events received from other agents. This implies loss of such events and their correlation. If this loss is critical, then such correlations could be performed by two independent agents – both subscribing to the same set of events and independently performing their correlations. An example of this is the replicated execution of Failure Detection Agents as described below.

3.3 Architecture for Self-Monitoring and Recovery

Figure 4 shows the central components of this system for performing self-monitoring and recovery. In addition to the System Management Agent (SMA), recovery involves two other kinds of agents: *Failure Detection Agent* (FDA) and *Recovery Agent* (RA). Any monitoring agents can be used to perform these functionalities by adding required detectors and handlers. The System Management Agent installs a configuration for self-monitoring by dispatching Failure Detection Agents and Recovery Agents at various hosts. Monitoring agents, System Management Agents, Failure Detection Agents, and Recovery Agents all generate periodic *AgentAlive* heart-beat events. These are received by the Failure Detection Agents.

A Failure Detection Agent receives periodic heart-beat events from the monitoring agents as well as other kinds of failure events, such as database server failures and RMI failures. It also has configuration information, and its function is to process the heart-beat events and generate specific events to indicate either a complete failure of an agent or failures of its detectors. Two or more Failure Detector Agents execute at different nodes and monitor each other to make sure that the loss of a Failure Detection Agent is detected and a new agent is created in its place.

A failure event triggers recovery procedures that are implemented in its handler. The recovery action is executed by one of the Recovery Agents, as described below. A pair of recovery agents executing on different hosts subscribe to the failure events generated by the Failure Detection Agents. This pair works in primary-backup mode – i.e., only the agent in the primary mode initiates any required recovery action. Failure Detection Agents also monitor the Recovery Agent pair. The agent in the backup mode becomes the primary when an event indicating the primary agent's failure is received. The recovery agent performs the following kinds of recovery functions. On a detector failure at an agent, it tries to re-install that detector. On agent failure, it recreates the agent based on its most recent configuration information, and re-launches it to the target host. Before sending the agent, it makes sure that the target host, its agent server, and RMI registry are running. Otherwise it first tries to restart them. To recover a failed System Management Agent, it

recreates it with the most recent checkpointed configuration state.

3.4 Agent Level Self-Monitoring

Each agent is equipped with an *AgentAliveDetector*, which periodically checks the internal state of the agent and generates appropriate heart-beat *AgentAlive* events to indicate the health of the agent. An *AgentAlive* event contains a list of detectors which are functioning in the agent. This event also contains a *current configuration number*. Whenever an agent's configuration is changed with the addition or deletion of a detector, this number is incremented. This number is also incremented when an agent is re-launched on recovery. The purpose of this is two-fold: first to make sure that the subscribers of an *AgentAlive* message would note that the configuration has changed; second, it is used to ignore any failure events related to old configurations.

Associated with an *AgentAliveDetector* is a handler and a list of subscribers. Any agent can subscribe to *AgentAlive* events from other agents, subject to system-wide security policies. The frequency of heart-beat events is configurable by the administrator depending on factors such as the load on a host, system configuration of the host, alert level of system operation, and network traffic among others.

Each Failure Detection Agent subscribes to *AgentAlive* events from all agents in the system. If no such event is received from an agent over a pre-defined number of consecutive timeout periods, it generates an *AgentFailure* event. It keeps on generating such events until the agent is restarted and a heart-beat message is received, or the configuration information is altered to ignore that agent. When a heart-beat message is received, the Failure Detection Agent compares the list of detectors in the event with its configuration information. It generates an *AgentFailure* event if a detector is found to be missing.

3.5 Restoration of Event Subscriptions

When an agent is restarted at a host, it does not have any information about its subscriber agents. In our model, the subscribers are required to register themselves with the event publishers to get the events of interest. Therefore, when an agent failure is detected, we need to inform all of its subscribers of this failure event. To facilitate this, each *AgentAlive* event from an agent also contains the list of its current subscribers. On detecting an agent failure, the Failure Detection Agent sends the *AgentFailure* event to all of its subscribers.

Each agent maintains a list of events that it subscribes to from other agents. It also maintains a list of *outstanding-subscriptions*, containing the list of agents with whom it has not yet succeeded in registering its subscriptions. *EventSubscriptionThread* in an agent periodically attempts to register subscriptions with the agents in this list and remove them from the list on success.

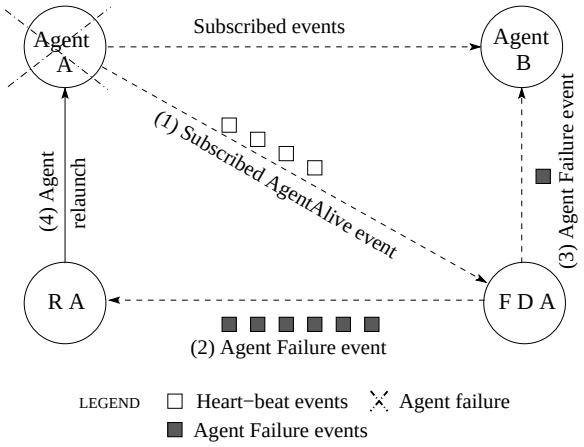

Figure 5. Recovery process of a Monitoring Agent

In Figure 5, the steps to recover and restart the failed monitor agent *A* are described. In the figure, agent *B* subscribes to events from agent *A*, Failure Detection Agent (FDA) subscribes to periodic *AgentAlive* events from agent *A*, and the Recovery Agent (RA) subscribes to *AgentFailure* events from FDA. As show in Step 1 of Figure 5, after predefined number of missed *AgentAlive* events, FDA generates and sends the *AgentFailure* event to the RA. This *AgentFailure* event continues to be generated periodically as long as the agent A is not recovered (Step 2 in Figure 5). Based on the subscription information of the earlier *AgentAlive* event, FDA also notifies all the subscribers (in this example agent *B*) of the failed agent *A*. On receiving an *AgentFailure* event, a subscriber agent puts the name of the failed agent in its *outstanding-subscription* list (Step 3). The *EventSusbcriptionThread* in the subscriber agent (in this example agent *B*) periodically tries to connect with the failed agent (in this example agent *A*) to register its subscription.

A Recovery Agent implements recovery procedure in the handler associated with the *AgentFailure* event. Upon receiving an *AgentFailure* event, the handler checks the status of the agent server of the failed agent by querying the agent server. If the agent server is alive, a new agent is created with the most recent checkpointed configuration of the failed agent and transfered to the agent server at its target host. Once the failed agent has been restarted at its host (Step 4), it registers its subscriptions to events from other agents.

3.6 Recovery of Itinerant Agents

Inspector agents are more prone to failures as they travel across the network. In a file integrity checker agent, a configurable timer is started at the launch servers for the agent, and on timeout a new inspector agent is launched. If the previous agent comes back it is terminated. Since actions of these inspector agents are idempotent by nature, multiple agents executing concurrently do not affect the integrity of the system nor the consistency of the data collected. If a host or agent server in the itinerary is down, the inspector agent automatically chooses the next host on the itinerary and reports the change to the recovery manager. Status reporting by inspectors provides valuable feedback about the state of the system. One can also verify the agent transfer protocol by using a *Ping Agent* that goes to the target host and returns.

3.7 System Performance

We have been running the Konark system more than a year in our lab environment consisting of 15-20 hosts. On average, the agent servers consume less than 1% of the CPU on SunBlade 100 workstations, running on a 502 MHz Sparc processor, and having 512 MB memory. The agent servers consume about 27 MB of memory, whereas a barebone JVM uses about 9 MB with default JDK 1.4 settings. With the period of heart-beat *AgentAlive* events set to 20 seconds and the number of timeouts set to 5, the recovery of a failed agent takes less than 2 minutes.

4 Security of the Monitoring System

Some components of the monitoring system need to run in an untrusted domain. The attackers might try to disable these components so that their activities are not detected. Hence, it is important for the monitoring system to be able to protect itself against attacks. It should be noted that some of the security requirements in our system are a direct result of the capabilities that we want to support. For example, if dynamic extensibility is not supported, then we don't have to worry about an attacker remotely modifying agent behavior.

4.1 Security of Agent Servers

An agent server running at a trusted host is assumed to be secure and trusted to execute agents "correctly", i.e., for example follow the agent transfer protocol for agent migration. An attacker could launch a malicious agent to an agent server and try to corrupt the server's data. This is prevented by using a distinct *protection domain* for each agent execution [6]. Agent servers also enforce restrictions on an agent's access to local resources, such as disk space, network ports, and files, based on its codebase and the principal on whose behalf it is acting.

Even though an agent server can enforce high-level resource access policies, it cannot control low-level resource usage such as the amount of CPU consumed. Hence, an attacker might try to launch a malicious agent which consumes too many host resources, thereby causing a denial-of-service attack. To prevent this, agent servers enforce an admission policy accepting agents coming only from a specified set of hosts and belonging to the system administrators. An agent server can also impose a limit on the number of agents that it might host at any time from a particular entity. These policies can be changed dynamically.

A malicious agent cannot tamper with other agents executing on an agent server, as each agent is executed in a *distinct protection domain* by its host agent server. A malicious agent might try to forge its identity. In Ajanta, this is prevented by giving each agent a set of *unforgeable credentials*. An attacker could also try to direct an agent away from an agent server thereby leaving the host unmonitored. An agent server prevents this by allowing these actions to be performed only by privileged entities such as an agent's owner or creator [6].

4.2 Security of Monitor Agents

There are many ways in which an attacker can try to exploit the capabilities that we support in our monitoring system. An attacker might try to mount attacks on a monitor agent by modifying its behavior by deleting or modifying existing detectors, or adding new detectors. An agent enforces policies, as specified by its owner, as to who can modify its behavior.

An attacker can also try to subscribe to events from an agent, thereby breaching confidentiality of sensitive event data. To prevent this, each monitor agent maintains a list of authorized subscribers. An attacker can also try to delete a valid subscriber registered with an agent and thus disrupt the monitoring system's operations. To prevent this kind of attack, in our system an existing subscriber can remove only its own subscription from an agent. An attacker can try to send events to subscriber agents thereby generating false alarms. To prevent this, every subscriber agent maintains a list of agents authorized to send events to it. The protection mechanisms to prevent these attacks require authenticated inter-agent communication. The solution used in this protocol requires an agent to trust a remote agent's server. The details of the authentication protocol are omitted.

4.3 Agent Security Policies

As dynamic extensibility is one of our main goals, the above policies might need to be modified during system operation. The agents allow these policies to be modified either by the administrators or other authorized users. The access control lists for an agent are constructed by SMA based on the configuration specified by the administrator. Access control is enforced on operations for adding, deleting, and modifying detectors, and for registering event subscriptions. The administrator has the privileges to modify all aspects of an agent, including the access control lists.

5 Related Work

Several researchers have investigated use of agents in network management [2, 8]. The functionalities of our system share goals with other intrusion detection systems, such as AAFID [1], Emerald [9], and GrIDS [11]. However, only few systems, such as Emerald [9] and Bro [7], have addressed security issues in monitoring systems. Compared to these systems, our monitoring system can self-monitor and self-recover.

Our approach for building self-recovering monitoring system has utilized concepts that have been proposed and used by others in the past [3, 5, 10]. Similar to the system presented in [3], our design uses the notion of soft state and peer-to-peer component monitoring and recovery. Resiliency of critical components of Konark is based on process replication concepts, similar to those used in the NonStop operating system [5].

6 Conclusions

In this paper, we have presented self-monitoring and self-recovery mechanisms of Konark, a mobile agent based network monitoring system. The same set of mechanisms used for monitoring infrastructure resources is also used to monitor its own components. Therefore, self-monitoring and recovery mechanisms inherit all the capabilities of the monitoring system, such as flexibility, dynamic extensibility, active monitoring, online-correlation of data, and self protection using security mechanisms of the mobile agent platform. The system incorporates features such as component-level self-monitoring, persistent recovery, and soft state management. The data correlation and aggregation mechanisms of Konark can be used to detect more sophisticated and subtle failures such as abnormalities in the context of the whole system.

References

[1] J. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An Architecture for Intrusion Detection using Autonomous Agents. Technical Report Coast TR 98-05, Department of Computer Sciences, Purdue University, 1998.

[2] P. Bellavista, A. Corradi, and C. Stefanelli. An Open Secure Mobile Agent Framework for Systems Management. *Journal of Network and Systems Management (JNSM)*, 7(3):323–339, September 1999.

[3] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Bewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Symposium on Operating Systems Principles*, pages 78–91, 1997.

[4] C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile Agents: Are they a good idea? Technical report, IBM Research Division, T.J.Watson Research Center, March 1995. Available at URL `http://www.research.ibm.com/massdist/mobag.ps`.

[5] J.F.Barlett. A NonStop Kernel. In *Symposium on Operating Systems Principles*, volume 15, pages 22–29, December 1981.

[6] N. Karnik and A. Tripathi. Security in the Ajanta Mobile Agent System. *Software Practice and Experience*, 31(4):301–329, April 2001.

[7] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1999.

[8] R. Pinheiro, A. Poylisher, and H. Caldwell. Mobile Agents for Aggregation of Network Mangagement Data. In *1^{st} International Symposium on Agent Systems and Applications, and 3^{rd} International Symposium on Mobile Agents*, pages 130–140, October 1999.

[9] P. A. Porras and P. G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 20^{th} National Information Systems Security Conference*, pages 353–365, October 1997.

[10] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299 – 319, December 990.

[11] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS: A graph based intrusion detection system for large networks. In *Proceedings of the 19^{th} National Information Systems Security Conference*, pages 361–370. National Institute of Standards and Technology, October 1996.

[12] A. R. Tripathi, M. Koka, S. Karanth, A. Pathak, and T. Ahmed. Secure Multi-Agent Coordination in a Network Monitoring System. In *Software Engineering for Large-Scale Multi-Agent Systems, 2002 (SELMAS 2002), Springer, LNCS 2603*, pages 251– 266, 2003.