

ROBUSTNESS BY SELF-MONITORING IN KONARK,
A MOBILE AGENT BASED NETWORK MONITORING SYSTEM

PLAN - B REPORT
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

HARSHA TALKAD

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

December 2004

UNIVERSITY OF MINNESOTA

This is to certify that I have examined
this copy of a Masters Plan B report by

Harsha Talkad

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Advisor : Prof. Anand Tripathi

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Abstract

A salient feature of any successful Distributed System is easy manageability to provide operability to a user in such a way that he/she is not concerned with the system health. Konark is a distributed network monitoring system built using the mobile-agent programming platform Ajanta. It is comprised of mobile-agents that can communicate with each other to perform system-wide correlation of data for management and protection of network computing environment. This project mainly deals with the issues related with robustness of Konark and its implementation.

To minimize management efforts, the system architecture has been designed with mechanisms to detect and self-recover from internal failures in a decentralized and scalable form. Self-recovery is achieved by using the same mechanisms as those used for monitoring resources in the network. Self-monitoring of Konark also has all the features of network monitoring such as dynamic extensibility, active monitoring and online-correlation of data. This work demonstrates the capability of a programming model using which distributed applications capable of self-monitoring and recovering from failures can be built.

Acknowledgments

I would like to thank Prof. Anand Tripathi for his constant encouragement, support and guidance throughout the course of this project. I would also like to thank students of Distributed Systems Lab - Tanvir Ahmed, Muralidhar Koka, Ivan Osipkov, Sandeep Karanth and David Johnson for their cooperation during the course of this Plan B project and during my association with the Lab and Konark project.

Contents

Chapter 1	Introduction	1
Chapter 2	Konark Overview	3
2.1	Motivation for Mobile-Agent Based Approach	3
2.2	Konark System Components	5
2.3	Monitor and Itinerary Agents	5
2.4	Operational Capabilities	7
Chapter 3	Robustness	8
3.1	Failure Modes of System Components	8
3.2	Goals and Requirements for Designing Self-Monitoring	10
3.3	Architecture of Self-Monitoring and Recovery	10
3.3.1	Definitions of Components	10
3.3.2	Architecture of Recovery System	13
3.4	Failure and Recovery Process	15
3.4.1	Restoration of Event Subscriptions	16
3.4.2	Failure and Recovery of SMA	16
3.4.3	Recovery of Itinerant Agents	17

3.4.4	Changes to Architecture	17
Chapter 4	System Capabilities and Performance	18
4.1	System Capabilities	18
4.2	System Performance	19
Chapter 5	Conclusions and Future Work	20
5.1	Conclusions	20
5.2	Future Work	21
Appendix A	Configuration File	22

Chapter 1

Introduction

Monitoring of computing system plays a major role in ensuring proper functioning of system services, detection of inconsistencies in system configuration and prevention of resource misuses. Also, it is important to monitor different aspects of a computing environment, such as network traffic, host level activities, system configuration, file systems, and user activities. Managing and monitoring large networks in any organization is becoming increasingly complex due to many factors like number of resources to be monitored, frequent addition of hardware and software components, huge amount of data being collected from diverse sources and the heterogeneity of the infrastructure. Monitoring systems need to inter-operate and correlate data across infrastructure with diverse technologies and policies. They need to cope with ever changing attacker's goals and their increasing abilities supported by sophisticated attack tools. They also need to secure themselves against attacks.

Mobile agent paradigm has been identified as a natural solution to implement this type of monitoring systems. In mobile agent based monitoring, autonomous software agents migrate to remote hosts and cooperate among themselves for system wide monitoring. Our mobile agent based monitoring system Konark [2, 3, 4] is based on this philosophy. Some of the design goals behind Konark are dynamic configurability, dynamic extensibility, active monitoring, security of monitoring infrastructure, robustness, scalability and acceptable system performance.

Due to above reasons, robustness of a monitoring system becomes an important issue. The Plan-B work concentrates on looking into the design and implementation of

robustness in Konark. Administrative efforts should be minimum in managing and configuring a monitoring system. One way to achieve this is by making the monitoring system resilient so that it performs on its own the recovery of failed components. The primary emphasis here is on repair and recovery of failed components to ensure continuous operation.

My project involved working on the various parts of the monitoring system. My specific contribution and the topic of this report is that of self-monitoring 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.

The structure of the report is as follows. Chapter 2 gives an overview of the network monitoring system addressing issues like how the system is built, its capabilities, architecture and design. The next chapter forms the core of the report. It starts with the problem, design goals. Follows it up with the architecture, design and working of self-monitoring component of Konark. Towards end of this chapter there are some experiments and numbers indicating the performance of the system and effect of self-monitoring on it. Chapter 4 briefly outlines the system capabilities and performance of existing system. Last chapter gives the conclusion and the future work and improvements. A small configuration file is attached as appendix to give an idea of what needs to be specified to achieve self-monitoring capability in current Konark implementation. Bibliography has list of references used for this project.

Chapter 2

Konark Overview

Before delving into the details of robustness, it will be useful to get an overview of design and working of Konark. This will help in visualizing the type of failures that can occur in the system. Also, as the systems monitoring capabilities have also been used to implement the robustness it is required to understand how monitoring works in the system for easy understanding of self-monitoring concept.

2.1 Motivation for Mobile-Agent Based Approach

The motivation behind mobile-agent based approaches is that it provides several capabilities such as local monitoring to overcome network latency and reduce network load, asynchronous execution, disconnected and autonomous operations, and dynamic adaptability [4]. A *mobile-agent* represents an object capable of migrating in a network to perform designated tasks at one or more nodes [5]. In our monitoring system, mobile-agents are sent to continuously monitor nodes in a network, perform data filtering locally, and notify other system components of any significant events. As mobile-agents are first-class objects, their state and behaviour can be altered remotely by invoking methods on them. These features are used for making the mobile-agents dynamically extensible and securely modifiable.

Most of today's monitoring systems rely on SNMP to collect data from various components. The SNMP model supports low level device management; it does not support abstractions for network-wide monitoring policies. In our approach, SNMP can be integrated as one of the building-blocks for low level device monitoring. Another com-

mon approach used in today's monitoring system is to periodically execute scripts to monitor the status of a component to process event data. Script based detection procedures tend to be tedious to install, debug and to modify remotely. In contrast to an object-based approach, scripts offer a lower level of abstraction for remote manipulation. This also has the disadvantage of a coarse-grain protection, as scripts execute with complete privileges of a specific user.

Konark [2, 3, 4], is network monitoring system implemented using Ajanta [5, 6], a secure Java-based framework for distributed programming with mobile agents. Ajanta framework comprises of three main components: agent, agent server and name registry. The framework facilitates creation of agents with primitives to control them, agent server hosts and provides resources to agents and location independent names are given to the global entities using the name registry. At lowest level, an agent is a Java object, which can be run by a thread, i.e., it implements the Java Runnable interface, and is sent from one agent server, i.e., a JVM to another. After the transfer, an agent is run as a separate thread starting one of the methods in the agent. Inter-agent communication in the monitoring system is based on the *publish-subscribe* paradigm implemented using RMI.

The design of Konark has been driven by the following goals.

Dynamic Configurability: The monitoring functions of an environment keeps on changing always due to software or hardware reconfigurations, updates/changes in administrative policies and improvement in monitoring tools. To accomodate all these constant changes the system should be dynamically configurable.

Dynamic Extensibility: Ability to add new monitoring mechanisms and event detection capabilities without bringing down the system or exetnding the functionalities of the existing components will be very useful feature to enhance the system's capabilities in response to new threats and policies.

Active Monitoring: Altering the detection policies in response to critical events is another desirable feature. This helps in starting additional monitoring functions and increase the level of alertness for specific events as and when some critical event is detected.

Robustness: Not allowing the attackers to bypass a monitoring system is one of the must-have requirement of any monitoring system. To achieve this, the

detection of component failures and restoration should be done with minimum or no human intervention which helps the recovery to be done in no-time.

Scalability and System Performance: The system should perform all the functions effectively with both small and large networks. Adding new nodes or networks to current monitoring domain should in no-way decrease the effectiveness of monitoring system. Also the system should not interfere with the normal functioning of the monitored hosts in terms of CPU and memory usage.

2.2 Konark System Components

As mentioned earlier publish-subscribe paradigm is followed in implementing Konark and an event model is followed where a *basic event* represents a significant change in the state of an entity being monitored. All the hosts/nodes in the environment to be monitored must run an agent server to support the installation and migration of agents. A typical configuration is shown in figure 2.1 and event detection is done through data collection, filtering and correlation.

System Management Agent(SMA) run on secure host is used to launch Monitor Agent(MA) based on configuration and remotely control and modify them. The 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 detection procedures and event subscription policies for event-correlation. Configuration database is used to maintain checkpoints and configuration changes, event database to store events for further correlation. System administrators interact with SMA through a management interface which also displays alarm events as soon as they occur for immediate action.

2.3 Monitor and Itinerary Agents

A Monitor Agent(MA) does the following. Execution of local detector functions, sending of event notifications and subscribing to events of interest generated by other agents. *Detectors* are the components that can be added or removed from an agent. They contain the functionality to be executed on any host and the type of application is dependent on what the detectors are programmed to do. Initially MA is launched with a variety of detectors for detecting basic events at its host as specified

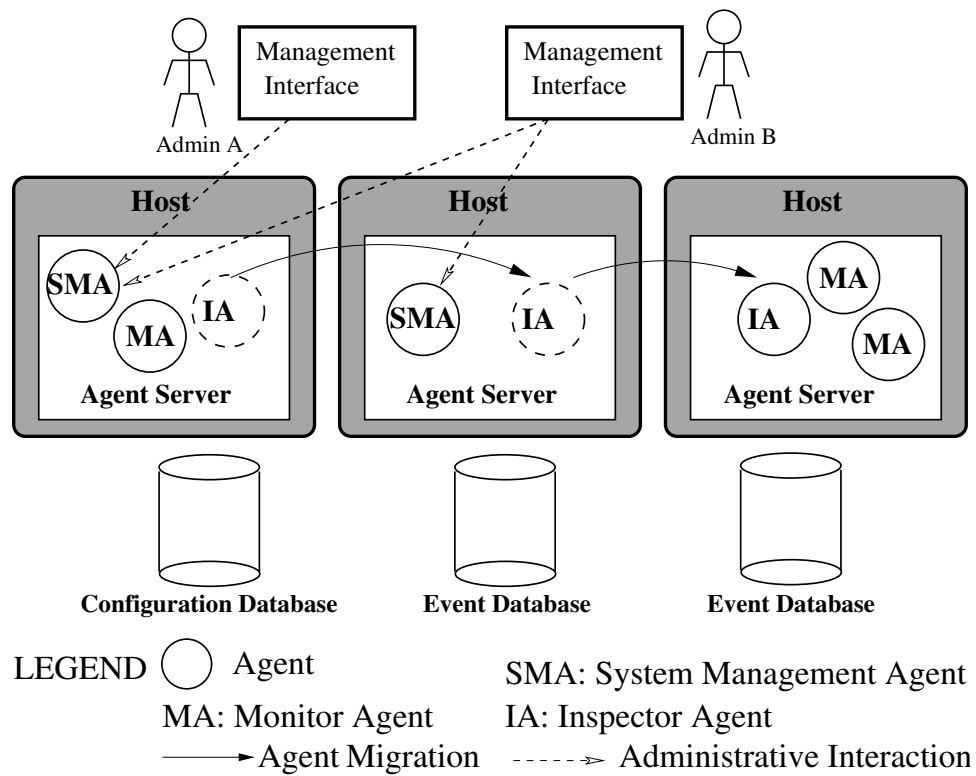


Figure 2.1: Monitoring system organization

in configuration. Each detector is triggered by events generated by other detectors. Associated with each detector is a handler which decides what should be done with the event generated. Each detector contains a thread which executes the detection procedure asynchronously with other activities in the agent. Both local events and events received from other agents are deposited in a single *event queue* which are processed in same manner. Depending on trigger table, dependent detectors are triggered. Depending on the subscribers, the event is sent to other agents.

Itinerary Agent is similar to monitoring agent in functionality but it travels to different hosts based on an itinerary, performing the specified detection procedures on each node it visits. Again the migration pattern, frequency of visits are configurable with a whole lot of options. This type of agents are used in Konark to execute same detection logic on many nodes with a lot of gap between the executions. This gives a single point of control and makes it easy to modify the agent so that effect of a single change is visible on many nodes.

2.4 Operational Capabilities

Some of the capabilities in current implementation are - processing different log files at host and trying to correlate the events, monitoring system file integrity(Trip Wire), fingerprint host configuration including typical services and routing table. For file integrity checker and fingerprinting *Itinerary Agents* are used as the type of logic that need to be executed at different host is same and it need to be executed once in a while. The itinerary agent carries the configuration information and execution logic to all the hosts at specified intervals. Network-wide events are correlated to find attacks, e.g., user's switch to multiple accounts, login from black-listed domains. Monitoring processing running with root privileges, runaway processes, failure of system level services, execution of malicious programs.

Chapter 3

Robustness

In the system explained in previous chapter there were many components whose failure and recovery needed to be addressed. Typically in current installation, each host runs at least one agent server with minimum of one agent always executing on it. On an average each agent has more than 20 detectors monitoring various aspects of the system. New detectors can be installed and also itinerary agents visit each host regularly. With so many components, one of the goal of the system was to reduce the burden on system administrators in configuring, managing and maintaining the system itself so that they can focus more on higher level policies and critical abnormal conditions.

The design was driven with goal of performing automatic recovery and repair of failed components within a short time-span so that any abnormal condition doesn't go unnoticed. Primary objective was not to find what went wrong but to ensure continued monitoring operations. Thus robustness was achieved by incorporating mechanisms [7, 8] for *self-monitoring* and *self-recovery* at different levels of the system architecture. The event detection, correlation, and notification mechanisms presented in previous chapter are used as basic building blocks.

3.1 Failure Modes of System Components

There are many types of failures that can hamper the Konark system from working properly. Some of these are not under the control of system and need human intervention to get things up and running again. Others can be handled by the system

and brought back to working condition. This section lists both types of failures and specifies which failures are handled by the system and which are not. The failures can be broadly categorized into following types.

1. **Host/Service failure** - is the failure of machine on which the agent server is hosting one or more agents. The failures due to crash of OS or some type of hardware failure definitely needs human intervention. But things like stopping of services like SSH, RMI registry necessary for Konark to run properly can be handled by the system itself. Because of security reasons the failures that need root access for correcting them are not handled by Konark (example restarting the SSH root daemon), but it definitely tries to bring up the processes/services that are running in user space(like RMI registry or Ajanta name registry).
2. **Agent server failure** - Agent server process can fail completely or partially. An agent server crash results in loss of all agents and their monitoring functionalities at the host, which requires restoring all agents that are permanently installed at the server. There can be partial failures where agents server is still running but can become inaccessible or may not accept agents because of failure of ATP(Ajanta Transfer Protocol).
3. **Agent failure** - An agent server can be hosting many agents. One or more agents can fail at a time. This requires only that particular agent to be recreated. A partial failure of agent can also occur when agents RMI interface may stop due to failure of rmi registry.
4. **Detector failure** - Each detector executes as a separate thread and any one of them can fail and stop performing its monitoring functions. A failed detector should be remotely replaced by a new detector by remote agent performing recovery actions.

The work presented here handles above mentioned failures except the ones requiring human intervention. In the following section the design and working of the failure and recovery architecture is described in detail.

3.2 Goals and Requirements for Designing Self-Monitoring

The goals and requirements that were kept in mind when designing the self-monitoring module of the Konark were as follows:

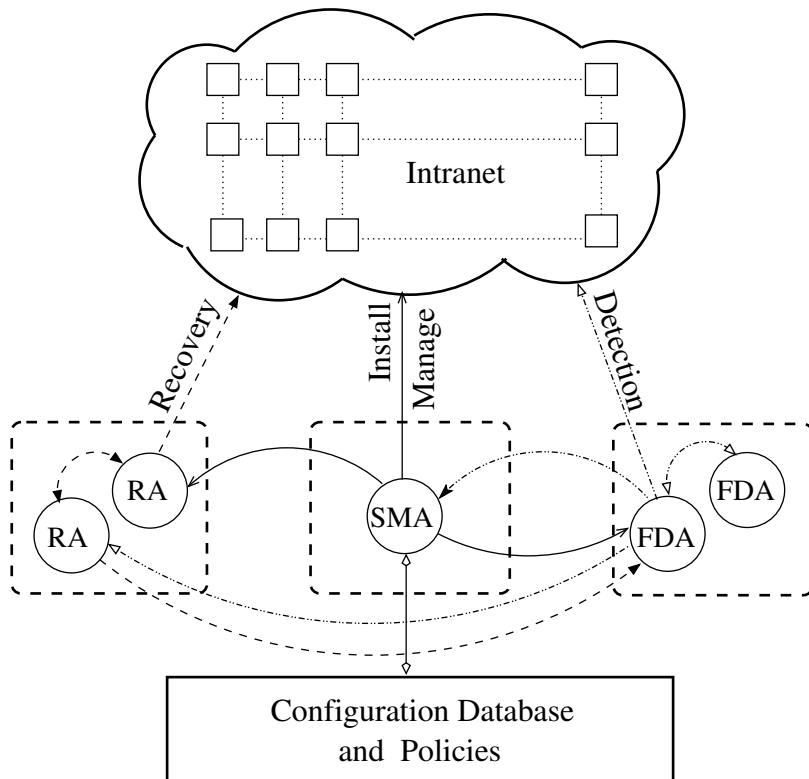
1. Minimize the time during which a particular resource remains un-monitored. This goal is the direct result of the main goal of Konark, which is monitoring the systems for any abnormalities. This is to reduce the possibility of an attack becoming successful because the attacker made an effort to disable the monitoring functionality so that he cannot be traced back to his origin.
2. Primary emphasis is on the repair and recovery of failed components rather than trying to find out what went wrong. Again the reason being fast recovery and monitoring rather than retrospection.
3. Using the system's inherent capabilities to detect its own component failures rather than coming up with the design of a new architecture for this purpose. This helps in keeping the design simple, easy to understand and maintain. This also uses the wonderful concept of using the capability to monitor systems to heal the system itself.
4. Maintaining or rather recreating the state of the system from the *soft-state* as much as possible instead of saving everything in a file or database. Since we are dealing with a distributed system this helps in reducing the problems introduced by redundant, inconsistent and/or stale data about the state of the system.

3.3 Architecture of Self-Monitoring and Recovery

Figure 3.1 shows the central components of this system for performing self-monitoring and recovery. Before explaining the architecture it is necessary to know the following definitions and this makes it easier to understand the architecture and implementation better.

3.3.1 Definitions of Components

1. **AgentAliveEvent Detector** - Detector that need to be present in an agent that needs to be monitored. This basically generates periodic heartbeat events



LEGEND

SMA: System Management Agent RA: Recovery Agent FDA: Failure Detector Agent

—→ Install & Manage - - - → Recovery ·····→ Detection

⌈ ⌋ Fault tolerant agent pair

Figure 3.1: Recovery Architecture

along with information required to find out about the health of the agent it is present in. The information also includes the current subscribers of the agent it is present in, which is later used in re-subscription when an agent is recreated.

2. **SMSAgentAliveEvent Detector** - Similar to agentalive events on a normal agent but for SMSAgent so that failure detector can know how to check if it is alive or not.
3. **FailureEvent Detector(FD)** - Detector with the logic of deciding whether an agent/detector has failed and needs to be repaired or reinstalled. The triggering event for this detector is AgentAlive Event. This expects periodic heartbeat from the agents and generates a Failure Event in case of missing heartbeats for configured amount of time. There is no need to specify any subscription information for this detector, an agent with a FD automatically subscribes to all agents with the AgentAliveEvent Detector when a system configuration is launched initially.
4. **SMSFailureEvent Detector** - Derived from failure event detector. Monitors specifically SMA. This was necessary because the SMSAgent cannot be brought back to its original state only through the soft-state and requires some form of information to be stored in disk or database.
5. **Failure Detection Agent(FDA)** - A normal monitor agent with a FailureEvent Detector in it is called FDA. Any agent can be made a FDA by adding a FD at any point of time when the system is running. There should be at least a pair of these agents running in the system. They should be hosted on agent servers residing on separate hosts for the system to be completely recoverable on its own.
6. **RecoveryHandler Detector**- The functionality to recreate a failed agent is implemented in this detector. Though this detector can be in any agent it is usually present in SMA for the following reason. All the latest information about any agent like the detectors in it, the subscribers to it or to whom the agent has subscribed for what, are present in the SMA as it is used to manage the entire system. In case of failure of any agent it has to be recreated and re-launched. The most natural place to do this is at SMA. So the RecoveryHandler Detector is usually in an SMA. This can also be added in other normal agents

but have to design a mechanism to provide it with the information present at SMA.

7. **SMSRecoveryHandler Detector** - Derived from recovery handler which implements the same functionality as the initial system launching but doesn't create all the agents, only the SMA. This is where the concept of *checkpoint* is important. SMA is used to store information about the entire system and manage it, but all the information about SMA is not stored in any other place. This problem is solved using check pointing, where the configuration is written to disk whenever there is a change in the configuration. The structure of check pointed file is kept similar to initial configuration file. So when the SMA fails only the checkpoint file is needed to recreate it.
8. **SMSRecovery Agent(RA)** - This is the agent with the SMSRecoveryHandler Detector. Other than this it is similar to any other monitor agent.

3.3.2 Architecture of Recovery System

Following are the *assumptions* made for the designed architecture to work. Any system configuration always has one SMA, pair of FDA and a RA. These four agents are hosted by agent servers running on different hosts. The chances of all the four hosts crashing/failing simultaneously is extremely low. To put it in other words at any given time the following pair of agents are healthy and performing their functions properly.

1. SMA and at least one FDA
2. RA and at least one FDA

A configuration file is added as appendix in the end which gives a simple configuration to demonstrate all the capabilities of the system.

Each agent is equipped with an AgentAliveEvent Detector, 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. Associated with an AgentAliveEvent Detector 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 the 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. Also it keeps track of failures using a sequence numbers assigned to each failure event. The sequence number is incremented for a new failure event and it wraps around after a long time. This sequence number helps in identifying duplicate failure events and prevents re-creation of a failed agent more than one time.

All the configuration information is necessary for recreating a monitoring agent when it fails and the natural place to find it is at the SMA which is used to manage the entire system. So the recovery of normal agents responsibility is given to SMA in the form of a detector. But this detector can be installed on any agent and given access to the required information to recreate the agent. Then question arises what happens when SMA fails?, Who can recreate the SMA and who has the information of the entire configuration so that it can be created. This problems was handled by using the persistent storage and files. One of the functionalities in SMA is *check pointing* itself, i.e., whenever there is a significant change in the configuration a checkpoint file is written onto disk. This has been implemented using call backs. The change to system is done by the user/administrator and this is done by using the **Admin GUI**. Call backs have been implemented so that depending on the actions done by the user in GUI function is called to checkpoint the changes in a file on disk. For further changes the same file is over written with new changes. The structure of the check pointed file is kept similar to the configuration file that is used to specify the initial system configuration. So when SMA fails, the SMA recovery detector uses the check pointed file to create the SMA and initializes necessary data structures so that once it is up it can again start managing the entire configuration.

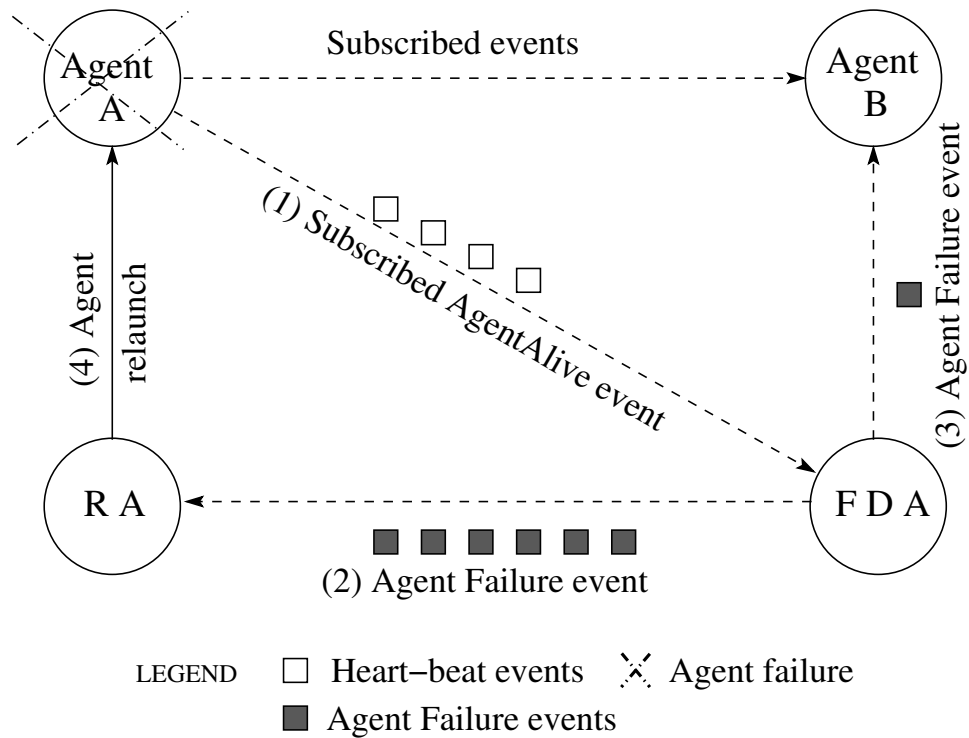


Figure 3.2: Recovery Process of Monitoring Agent

3.4 Failure and Recovery Process

The figure explains the steps to recover and restart the failed monitor agent A. In the figure, agent B subscribes to events from agent A, Failure Detection Agent subscribes to periodic AgentAlive events from agent A, and the Recovery Agent(RA) subscribes to AgentFailure events from FDA. As shown in Step 1 of Figure 3.2 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 3.2). 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 eventsubscription thread in the subscriber agent(in this example agent B) periodically tries to connect with the failed agent(in this example 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 transferred to the agent server at its target host. Once the failed agent has been restarted at its host, it registers its subscriptions to events from other agents.

3.4.1 Restoration of Event Subscriptions

Once an agent is recreated the question that arises is how to inform all the agents who were initially subscribed to the failed agent for any events. There is no way for them to know that this particular agent has failed and they need to re-subscribe. It is hard to detect this based on just arrival or non-arrival of events because non-arrival may be due to non generation of events. So to overcome this problem the following strategy was followed.

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. The subscribers then puts this in the Outstanding-subscription list. A separate thread works on this list and tries to re-subscribe at frequent intervals. The retries goes on as long as it is either successful in subscribing or it is stopped from re-subscribing.

3.4.2 Failure and Recovery of SMA

As explained earlier SMA(System Management Agent) is the agent which is used to launch, control and modify the entire system. This is the place where information about the entire system is present. It would be very difficult if not impossible to recreate all these information in case of failure through the soft-state concept. So to keep the design and recovery process simple *check pointing* is used to help recover a failed SMA. A checkpoint file is specified during the launching of the system on a secondary storage. Then the system through the call back functions implemented in GUI, starts storing all the changes in the state of the system. The format of the

check pointed file is kept same as the initial configuration file so that recreation will be nothing but re-launching of the system without recreation of agents.

3.4.3 Recovery of Itinerant Agents

Itinerary agents are more prone to failures as they travel across the network. To handle this a configurable timer is started at the launch servers of the agent, and on timeout a new itinerary agent is launched. If the previous agent comes back it is terminated. Since the actions of the agents and detectors have been implemented to be 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 itinerary agent automatically chooses the next host on the itinerary and reports the change to the recovery manager.

3.4.4 Changes to Architecture

When implementing this self-monitoring and self-recovery functionality in Konark system there were some changes that were required to be made to the architecture of the system itself. These turned out to be good design changes which later became useful in implementing other monitoring capabilities in system.

Local and Remote Events - Initially we never distinguished between local or remote events of the same type and there was no restriction on where an event can be sent. As a result, the following situation surfaced while implementing the Failure Detection Agent pair. Each of these agents periodically generated AgentAlive Events and each subscribed to these events from all such agents, including itself. As a consequence, an AgentAlive event generated by one of the agents would be bounced back and forth between the agents creating an infinite loop. Besides that, locally generated AgentAlive events would end up being locally consumed, i.e., a self-subscription was in effect, which we found wasteful as it had no useful purpose. These observations led us to enforce the following general rule in our framework. An event is never sent to an agent that originally created that event. As a result of this we also implemented a way of distinguishing between remote and local events in defining trigger relationships. This gave more control in creating the configuration of the system.

Chapter 4

System Capabilities and Performance

4.1 System Capabilities

This monitoring system can be used to process data from a wide-range of sources. In the current implementation, at a host it processes log files, monitors system files for integrity and fingerprint(host configuration, routing table etc), and monitor processes running on it. This monitoring system can also be integrated with other tools and COTS(Components Off The Shelf) easily. An agent can be used as a wrapper to process the information generated by these tools, thus generating events which can be correlated with other events in the network. Currently the external tools that are used in similar mode are snort to detect network-level attacks and kismet a wireless sniffer to find out mac address spoofing. Users can build basic and compound event detectors based on data from these diverse sources.

Though most of the experiments that have been conducted largely focused on development of application for monitoring Network and ways to manage and recover from failures, the same capabilities can be used in a development of variety of distributed applications. The only requirement is to model the distributed application in terms of event detection and correlation and rest of framework with self-monitoring and recovering capability is already present.

4.2 System Performance

We have been running the Konark system for more than a year in our lab environment consisting of 15-20 hosts. On an average, the agent server consumes less than 1% of the CPU on SunBlade 100 workstations, running on a 502 MHz Sparc processor, and having 512MB memory. The agent server consume about 27MB of memory, where as a bare bone JVM uses about 9MB 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.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This approach for building self-recovering monitoring systems has utilized concepts that have been proposed and used by others in past. This design uses the notion of soft state and peer-to-peer component monitoring and recovery. The same set of mechanisms used for monitoring system, such as flexibility, dynamic extensibility, active monitoring and online-correlation of data. 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.

The design of Konark can be used as a framework for building event-based distributed applications. It provides mechanisms for detection, notification and handling of events collected at different locations in a distributed systems. It also provides framework for integrating mechanisms for self-monitoring. The work here illustrates the capabilities of autonomic recovery in a test bed system for monitoring network computing environments which can be extended in any distributed application based on this framework.

5.2 Future Work

Some of the improvements that can be done in the autonomic recovery implementation explained in this report are

- 1 - Suspension of functionalities, the threads and migration of the agent with all the state information to a different hosts and restarting with same process execution environment. By implementing this it will be easier to move the agents from one host to another without losing any information. This will be useful for load balancing or when a machine need to be brought down for maintenance or upgrade.
- 2 - Changing the current configuration file based specification of monitoring to policy based monitoring and get rid of static information regarding to achieve autonomic recovery. This will reduce the current size of configuration file considerably and will be less prone to errors.
- 3 - Some Broker type of service where agents can register the type of events they are generating and also type of events they need in terms of attributes of the events as opposed to current implementation where both the agent and event name is needed to subscribe to an event.

Appendix A

Configuration File

```
SYSTEM_VALUE{
    DEFAULT_HANDLER = network.manager.Defaultmanager.EventHandler;
    TIMER_FREQ = 20000;
    CPU_TIMER_FREQ = 60000;
    SYSLOG_FILE = /var/adm/messages;
    IPE_FILE = /project/space03/konark/develsrc/network/config/ipepolicyfile;
    ARM_ARGS_FILE = /project/space03/konark/develsrc/network/config/armargs;
    DAEMONS_FILE = /project/space03/konark/develsrc/network/config/daemons;
    PATTERN_FILE = /project/space03/konark/develsrc/network/config/SyslogPatterns;
    DOMAIN_HOSTS_FILE = /project/space03/konark/develsrc/network/config/hosts;
    MONITORUSERS_FILE = /project/space03/konark/develsrc/network/config/logUsers;
    ROOTLOGIN_HOSTS_FILE = /project/space03/konark/develsrc/network/config/roothosts;
    VALID_USERS_FILE = /project/space03/konark/develsrc/network/config/validUsers;
    CHECK_POINT_DIR = /home/grad00/konark3/konark/backup;
    CHECK_POINT_FILE = cp;
    SMS_AGENT_NAME = ia;
    SMS_AGENT_URL = /copernicus.cs.umn.edu:20000/URN:
                    ans:archimedes.cs.umn.edu/konark3/smsAgent;
}

SMS_AGENT{
```

```

TARGET_HOST = URN:ans:archimedes.cs.umn.edu/konark3/copernicus.cs.umn.edu;
AGENT_NAME = ~/ia;
dburl = jdbc:mysql://archimedes.cs.umn.edu:10000/test;
dbuser = mobile_agent;
dbpasswd = user1000;
TRIGGER_TABLE = /home/grad00/konark3/konark/trigger;
DETECTOR{
    network.detectors.TimerEventDetector, network.manager.EventHandler;
    network.detectors.SMSAgentAliveEventDetector, network.manager.EventHandler;
    network.detectors.RecoveryHandlerDetector, \
        network.manager.RecoveryHandler, "HandlerOnly";
}
}

```

```

AGENT{
    TARGET_HOST = URN:ans:archimedes.cs.umn.edu/konark3/failureAgent1;
    AGENT_NAME = ~/failureAgent1;
    dburl = jdbc:mysql://archimedes.cs.umn.edu:10000/test;
    dbuser = mobile_agent;
    dbpasswd = user1000;
    TRIGGER_TABLE = /home/grad00/konark3/konark/trigger;
    DETECTOR{
        network.detectors.TimerEventDetector, network.manager.EventHandler;
        network.detectors.AgentAliveEventDetector, \
            network.manager.AgentAliveEventHandler;
        network.detectors.FailureEventDetector, network.manager.RemoteEventHandler;
        network.detectors.SMSFailureEventDetector, network.manager.EventHandler;
    }
    SUBSCRIPTION{
        AGENT_NAME = ~/ia;
        EVENT{
            network.events.SMSAgentAliveEvent, network.manager.EventHandler;
        }
    }
}

```

```
}
```

```
AGENT{
```

```
    TARGET_HOST = URN:ans:archimedes.cs.umn.edu/konark3/failureAgent2;
```

```
    AGENT_NAME = ~/failureAgent2;
```

```
    dburl = jdbc:mysql://archimedes.cs.umn.edu:10000/test;
```

```
    dbuser = mobile_agent;
```

```
    dbpasswd = user1000;
```

```
    TRIGGER_TABLE = /home/grad00/konark3/konark/trigger;
```

```
    DETECTOR{
```

```
        network.detectors.TimerEventDetector, network.manager.EventHandler;
```

```
        network.detectors.AgentAliveEventDetector, \
```

```
            network.manager.AgentAliveEventHandler;
```

```
        network.detectors.FailureEventDetector, network.manager.RemoteEventHandler;
```

```
    }
```

```
}
```

```
AGENT{
```

```
    TARGET_HOST = URN:ans:archimedes.cs.umn.edu/konark3/recoveryAgent;
```

```
    AGENT_NAME = ~/recoveryAgent;
```

```
    dburl = jdbc:mysql://archimedes.cs.umn.edu:10000/test;
```

```
    dbuser = mobile_agent;
```

```
    dbpasswd = user1000;
```

```
    TRIGGER_TABLE = /home/grad00/konark3/konark/trigger;
```

```
    DETECTOR{
```

```
        network.detectors.TimerEventDetector, network.manager.EventHandler;
```

```
        network.detectors.AgentAliveEventDetector, \
```

```
            network.manager.AgentAliveEventHandler;
```

```
        network.detectors.SMSRecoveryHandlerDetector, \
```

```
            network.manager.SMSRecoveryHandler, "HandlerOnly";
```

```
    }
```

```
SUBSCRIPTION{
```

```
    AGENT_NAME = ~/failureAgent1;
```

```
    EVENT{
```

```

        network.events.SMSFailureEvent, network.manager.EventHandler;
    }
}

AGENT{
    TARGET_HOST = URN:ans:archimedes.cs.umn.edu/konark3/dec.cs.umn.edu;
    AGENT_NAME = ~/dec.cs.umn.edu;
    dburl = jdbc:mysql://archimedes.cs.umn.edu:10000/test;
    dbuser = mobile_agent;
    dbpasswd = user1000;
    TRIGGER_TABLE = /home/grad00/konark3/konark/trigger;
    DETECTOR{
        network.detectors.TimerEventDetector, network.manager.EventHandler;
        .
        .
    }
    SUBSCRIPTION{
        .
        .
    }
}

AGENT N-1{
    .
    .
}

AGENT N{
    .
    .
}

```

Bibliography

- [1] A. Tripathi, M. Koka, S. Karanth, I. Osipkov, H. Talkad, T. Ahmed, D. Johnson and S.Dier Robustness and Security in a Mobile-Agent based Network Monitoring System In: Appeared in International Conference on Autonomic Computing (2004)
- [2] Tripathi, A., Koka, M., Karanth, S., Pathak, A., Ahmed, T.: Secure multi-agent coordination in a network monitoring system. In: To appear in, Software Engineering for Large-Scale Multi-Agent Systems, Springer, LNCS #2603 (2003)
- [3] Tripathi, A., Ahmed, T., Pathak, S., Pathak, A., Carney, M., Koka, M., Dokas, P.: Active Monitoring of Network Systems using Mobile Agents. In: Networks 2002, a joint conference of ICWLHN and ICN 2002. (2002) 269–280
- [4] Tripathi, A., Ahmed, T., Pathak, S., Carney, M., Dokas, P.: Paradigms for Mobile Agent-Based Active Monitoring of Network Systems. Technical report, Department of Computer Science, University of Minnesota (2001) Available at URL <http://www.cs.umn.edu/Ajanta>.
- [5] Tripathi, A., Karnik, N., Vora, M., Ahmed, T., Singh, R.: Mobile Agent Programming in Ajanta. In: Proceedings of the 19th International Conference on Distributed Computing Systems. (1999) 190–197
- [6] Karnik, N., Tripathi, A.: Security in the Ajanta Mobile Agent System. *Software Practice and Experience* **31** (2001) 301–329
- [7] G. Candea, P. Keyani, E. Kiciman, S. Zhang, and A. Fox. JAGR: An Autonomous Self-Recovering Application Server. In *Autonomic Computing Workshop*, pages 168–177, June 2003.

- [8] N. H. Minsky. On Conditions for Self-Healing in Distributed Software Systems.
In *Autonomic Computing Workshop*, pages 86–92, June 2003.