
Autonomic configuration and recovery in a mobile agent-based distributed event monitoring system



Anand R. Tripathi^{*†}, Devdatta Kulkarni, Harsha Talkad, Muralidhar Koka, Sandeep Karanth, Tanvir Ahmed and Ivan Osipkov

Department of Computer Science, University of Minnesota, Twin Cities, MN 55455, U.S.A.

SUMMARY

In this paper we present a framework for building policy-based autonomic distributed agent systems. The autonomic mechanisms of configuration and recovery are supported through a distributed event processing model and a set of policy enforcement mechanisms embedded in an agent framework. Policies are event-driven rules derived from the system's functional and non-functional requirements. Agents in the network monitor the system state for policy violation conditions, generate appropriate events, and communicate them to other agents for cooperative filtering, aggregation, and handling. A set of agents perform policy enforcement actions whenever events signifying any policy violation conditions occur. Policies are defined using a specification framework based on XML. The policy enforcement agents interpret the policies given in XML. We illustrate the utility of this framework in the context of an agent-based distributed network monitoring application. We also present an experimental evaluation of our approach. Copyright © 2006 John Wiley & Sons, Ltd.

Received 29 December 2005; Revised 23 June 2006; Accepted 29 June 2006

KEY WORDS: agent-based systems; distributed event processing; autonomic systems; policy-based system management

1. INTRODUCTION

The management of large-scale software systems to ensure resilient operations is a challenging task. We present here an approach for the autonomic management [1] of large-scale agent-based distributed

*Correspondence to: Anand R. Tripathi, Department of Computer Science, University of Minnesota, Twin Cities, MN 55455, U.S.A.

†E-mail: tripathi@cs.umn.edu

Contract/grant sponsor: US National Science Foundation; contract/grant number: 0087514

systems, based on policy-driven mechanisms for configuration and recovery. The approach is presented in the form of a framework, called Konark [2,3], which is a multi-agent system for programming distributed event-based applications. The framework provides essential mechanisms for policy-based autonomic management of agent-based distributed systems. This policy-centric approach evolved through our experiences in building and experimenting with an agent-based distributed system for event monitoring in network computing environments.

Our approach is based on the agent-based integration of components and services in building distributed systems. Agents are the basic building blocks in our approach as they provide an ideal foundation for policy-based component integration. An agent is an active object encapsulating other components [4,5]. It acts as a container for its embedded components and possesses certain security privileges. Typically, an agent has certain functional role—it may work on behalf of a user requesting services from other agents, or it may act as a service provider. A service may be implemented by a group of cooperating agents. Moreover, agents and components may be transportable in the network, thus providing additional capabilities for their remote installation.

The policy-based mechanisms presented here address the need for autonomic operations in a wide class of distributed systems that are based on service-oriented architectures. Examples of such systems include distributed monitoring systems, pervasive computing environments, and Internet-based infrastructures and services. Such systems operate as a federation of a large number of distributed components and services. These systems also tend to be dynamic: new components or services are frequently integrated into the system, or existing components or services are removed due to failures or administrative policies. Evolutionary growth is an essential feature of such systems, requiring the frequent installation of new services and components. The mobility of users and their computing devices is another aspect of the dynamic nature of such systems.

In managing large-scale systems, continuous intervention and supervision by human administrators is not feasible due to the scale of operations involved with a large number of distributed resources. Moreover, in a large system, component failures cannot be eliminated. Therefore, suitable mechanisms are needed for autonomic operations that support configuration management and recovery to deal with failure conditions.

In our approach, policies for autonomic management are derived from an application's functional and non-functional requirements. Policies represent constraints and requirements on the configuration of agents and the inter-agent interaction. Based on policies, enforcement rules are derived that embody the policy enforcement actions that are required to be executed when some significant events occur. Policies act as a glue in creating a dynamic configuration of the system in order to satisfy the functional and non-functional requirements. Monitoring of policy violations through the detection of *policy events* and performing *policy actions* to ensure that the application-level and system-level requirements are not violated forms the basic design principle in this approach.

We demonstrate the utility of this approach through the design of a system for monitoring network computing environments, which we implemented using the Konark framework [3,6]. A network monitoring system for a large computing environment has several requirements. The system should be dynamically extensible to support the installation of new monitoring functions at a host or modifications of existing functions. It should be possible to define any desired event subscription/notification relationships among distributed agents. These relationships may need to be established dynamically, based on the attributes of events and agents. These requirements are addressed through the policies for agent composition, agent interactions, agent failure monitoring, event detection, and subscription/notification of events among agents.

A distributed event monitoring application is programmed in Konark as a collection of cooperating mobile agents. A mobile agent represents an active object capable of migrating in the network to perform certain designated tasks at one or more nodes [7]. An agent typically contains a number of event detector and handler objects. The functionality of an agent can be dynamically altered by adding or removing its components in a secure manner. An agent provides a convenient mechanism to transport code to a host to autonomously perform a desired set of monitoring functions. Mobile agents are sent to continuously monitor nodes in a network, detect and generate events, execute event handlers to process events, and send event notifications to other agents. The distributed event processing model forms the basis for implementing the policy-based approach presented here for autonomic management.

There are four main contributions of this paper. First, we present an agent-based model for programming distributed event processing applications requiring the collection and correlation of distributed event streams. Second, we present policy-based approach for building autonomic mechanisms for configuration and recovery in distributed agent systems. The basic concepts of this approach were developed in [8]. Third, we present here an XML-based specification framework for expressing policies for a system. We demonstrate the utility of this approach through a network event monitoring application. Finally, we evaluate performance of these policy-based autonomic mechanisms and present design guidelines for their scalable implementation.

In Section 2 we present the details of the agent architecture and agent-based event processing model. They form the basic building blocks of policy-based autonomic management. In Section 3 we give an overview of the Konark-based network event monitoring application. In Section 4 we present examples of some of the policies used in the network monitoring system and describe the basic elements of the XML-based policy specification framework. Section 5 presents our approach of policy-based autonomic management in large-scale distributed agent systems. In Section 6 we evaluate the policy-based design of a network event monitoring application and perform a detailed scalability analysis of autonomic mechanisms for configuration and recovery in Konark.

2. AGENT-BASED DISTRIBUTED EVENT PROCESSING MODEL

In this section, we present the agent architecture and the agent-based distributed event processing model of the Konark framework. This framework also serves as the basis for implementing our policy-driven approach for autonomic management in agent-based systems.

2.1. Agent architecture

Figure 1 shows the architecture of an agent in the Konark framework. An agent contains three types of basic components: event detectors, event handlers, and event dispatchers. An agent provides a framework for dynamic integration of these components supporting an event-based execution model. It also provides *interfaces* for remotely installing components on the agent or establishing event subscription/notification relationship between agents.

The *Component Management Interface* of an agent allows other agents to remotely add, remove, or modify the detectors and handlers in the agent. Using the *Subscriber Interface*, remote agents register their interest in subscribing to certain types of events. An agent receives events from remote agents through the *Notifier* interface.

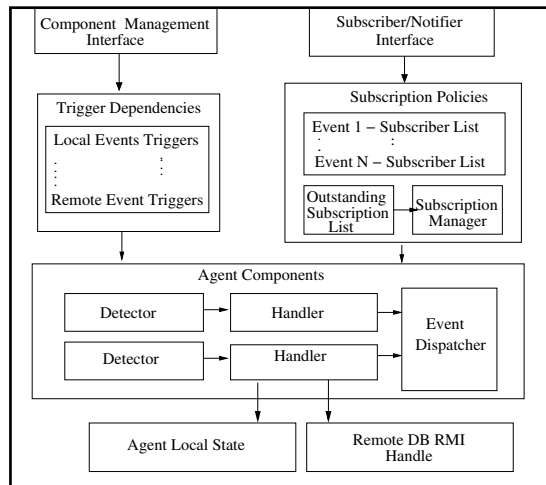


Figure 1. Agent architecture.

Interactions between agents are controlled by the *subscription policies* of the agents. The subscription policy of an agent is related to two kinds of requirements. The first specifies the agents who are allowed to subscribe to the events generated by this agent. The second specifies the events generated by other agents that this agent should subscribe to. An *Outstanding Subscription List* is maintained by each agent, listing other agents with whom it should register event subscriptions. The agent also contains a thread called *Subscription Manager*, whose function is to ensure that this agent's event subscriptions at other remote agents are properly established during initialization or recovery.

The agents in the Konark framework are programmed using Ajanta [9], which is a Java-based mobile agent programming system. The Ajanta system provides facilities to build customizable *servers* to host mobile agents, primitives for the creation and management of mobile agents, and a global naming service. Each agent in Ajanta is identified with a unique Ajanta defined Uniform Resource Name (URN) [10], which is a location-independent name. Inter-agent communication is based on RMI. Ajanta provides a *Name Registry* in which all the agents in the domain are registered. The Ajanta Name Registry provides APIs for mapping an agent's URN to the corresponding RMI URL. This facility is used for establishing event subscription/notification relationships between agents. Ajanta provides security mechanisms for authenticated inter-agent communication and enforcing policies for controlling admission of agents at servers [3]. The publish/subscribe model for event communication between agents builds upon the RMI-based inter-agent communication abstraction provided by Ajanta. Dynamic extensibility of applications through the installation of new detectors and handlers on agents is supported through the remote installation mechanisms provided by Ajanta.

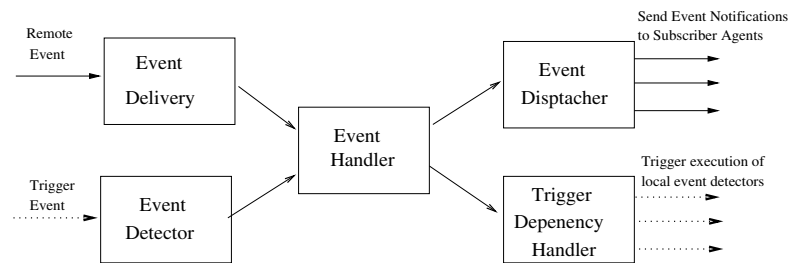


Figure 2. The event-based execution model.

2.2. Agent-based distributed event model

The event-based execution model for an agent is shown in Figure 2. The functions of the five stages involved in the processing of events within an agent are detailed below.

2.2.1. Event delivery

Events are delivered to an agent through its subscriber/notifier interface. This interface supports authenticated communication and it can be used to specify the agents who are authorized to send events to this agent, and the types of events that can be accepted from a given agent.

2.2.2. Event detectors

An agent is launched with a variety of event detectors for monitoring application-defined conditions. The detectors encapsulated in each agent have very specific tasks. They observe some system attribute and generate an event when the specified condition is detected. An event is a Java object, and events are related in a class hierarchy. A detector is invoked based on the occurrence of some specified types of events, which are called *triggering events* of that detector. Each event class definition specifies the triggering events for its detector object. Such relationships between detectors and event types are called *trigger dependencies*. A detector is implemented as a single-threaded object, which continuously waits for trigger events and executes its event detection function whenever a trigger event is delivered to it. The detector function is given the triggering event as an argument, thus a detector can perform some filtering operations on it to detect a subclass of the triggering event. For example, a *user login* event may trigger the *root login* event detector to simply check if the given login event corresponds to the super-user.

A detector in an agent can be triggered by a local event generated by another co-located detector, or by a remote event that the agent receives from another agent. The detector execution may result in the generation of one or more events of a given type.

2.2.3. *Event handlers*

In an agent, a handler object is associated with each event type. Multiple events of a particular type are handled sequentially by the event handler. When an event is detected or received by an agent, it is given to the handler. The handler object performs the required processing action, such as storing events in local or remote databases, sending alerts to system administrators, creating and launching new agents, and modifying an existing agent by installing or removing detector or handler objects. After the processing by the handler, the event is given to the *Trigger-dependency handler* and the *Event Dispatcher*.

It is possible for an agent to have only the handler object for an event and not to have any local detector for that event type. In such a case, the handler is used for processing the events that are detected and notified by remote agents. It is also possible for different agents to have different kinds of handlers for a given event type. A handler may also specialize its actions based on the attributes of the event being processed.

2.2.4. *Trigger-dependency handler*

After processing of an event by the appropriate handler in the agent, it is given to the trigger-dependency handler, which is responsible for triggering other local detectors according to the trigger dependencies. Trigger dependencies identify the triggering relationship between the event detectors. A detector can be triggered by both *local* and *remote* events. Local and remote events that trigger a detector are separately specified for each detector. The classification of the trigger events as *local* and *remote* is required when one needs to handle a given type of event differently, based on its local or remote origin. This separation also helps in ensuring that no useless processing of certain kinds of events is performed when they are local.

2.2.5. *Event dispatcher*

The event dispatcher sends the event to all the remote subscriber agents registered for that event type. Multiple events are handled sequentially by the event dispatcher. Each event type may have various policies associated with it, such as an event should not be sent back to its originator and an event should be disposed off after a certain number of hops. The event dispatcher evaluates such event policies before forwarding the events to the remote subscribers. Such event policies are essential to limit indefinite event circulation in the system and thus maintain system stability. It is also possible for the event dispatcher to determine at runtime the set of subscribers for an event based on its attributes.

3. DESIGN OF A NETWORK EVENT MONITORING SYSTEM USING KONARK

In this section, we present an overview of the network event monitoring system that we have developed using the Konark framework. The design of this network event monitoring system evolved through two design phases. Our initial design mainly concentrated on validating the use of mobile agents for performing network monitoring. Policy-based configuration and management was not the central goal

in this initial design. However, our subsequent design approach was completely based on policy-based configuration to address the shortcomings of our initial design.

In our design, one agent called the *System Management Agent* (SMA) is used for managing the system configuration, which includes creating, configuring, and launching agents according to system-level monitoring goals. A SMA typically runs on secure hosts and remotely controls agents deployed in the system. System administrators interact with the SMA through a GUI console tool. The SMA subscribes critical events from other agents and displays them on the GUI console. The high-level goals of this network event monitoring system are to monitor hosts in a network for various kinds of events. Examples of events include user logins, program executions on a host, file modifications, and network traffic patterns related to intrusions.

3.1. Detectors and handlers

Detectors deployed on an agent perform the host monitoring functions such as: processing log files at each host, verifying checksums of system files for integrity, fingerprinting host configuration including daemon services and routing table, monitoring processes running with root privileges, runaway processes or those consuming resources over predefined thresholds, failure of system-level services such as termination of daemons, and the execution of malicious programs such as packet sniffers and password crackers. Detectors generate events corresponding to the above activities. Events are related in a class hierarchy. A class higher in the hierarchy represents a general event type and its subclasses are events representing specific conditions within that general type. Events higher in the hierarchy are detected first and then passed to the detectors at the lower levels. This enables selective triggering of detectors at lower levels in the hierarchy. Figure 3 shows an example event hierarchy. The *SyslogEvent* class represents the events generated from the system log files. The *ConnectEvent* class represents various kinds of connections to a host. Its subclasses are events corresponding to SSH, FTP, etc. Thus the *LoginEvent* detector is triggered only when an event of type *ConnectEvent* is determined to have occurred, based on the filtering of *SyslogEvent*. It is also possible to perform a network-wide correlation of events to find attacks, such as abnormal root login activity in the domain, user's switching to multiple accounts, and logins from black-listed domains.

Figure 4 shows the trigger dependency for the *AbnormalRootLogin* detector. The primary function of this detector is to monitor system-wide abnormal root login activities. In each agent, a periodic timer event triggers the execution of the *SyslogEvent* detector. This detector generates events based on new log entries in the system log files. A *SyslogEvent* triggers executions of the *ConnectEvent* detector, which filters and generates any login related events. This event triggers detectors for specific kinds of login, such as SSH, Telnet, and rlogin. These detectors are filters that check if a given connection event belongs to a specific class. These events trigger execution of the local *RootLogin* event detector, which checks if a given login event corresponds to the super-user. The *RootLogin* events, both locally and remotely generated, trigger the *AbnormalRootLogin* detector. Typically, just one agent in the system is required to have this detector to monitor and correlate all *RootLogin* events in the environment. The function of this detector/agent is to monitor for any abnormal root login activities, such as a root login originating from a foreign domain or a large number of root login activities involving multiple hosts in a short period. In the trigger-dependency graph of Figure 4, this detector is marked to be triggered by both local and remote login events. A trigger dependency marked only as *local* implies that the triggering event detector must be co-located in the same agent with the triggered detector.

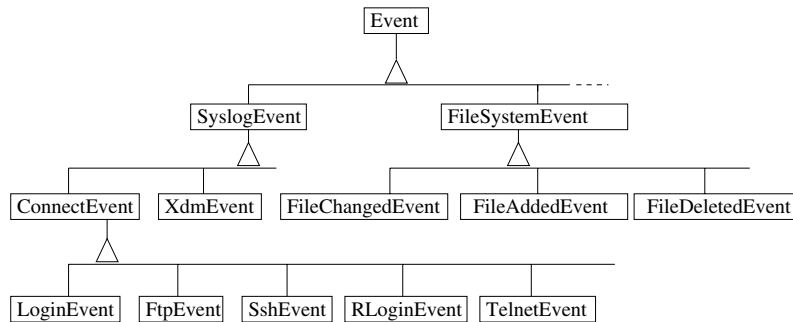


Figure 3. An example event hierarchy.

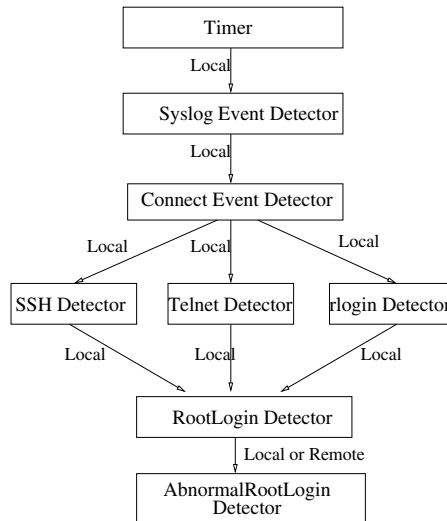


Figure 4. Trigger dependency—example 1.

Similarly, a dependency marked only as *remote* implies that the triggering event must originate from a remote agent.

Another example illustrating the distinction between the *local* and *remote* trigger dependencies is shown in Figure 5. To detect failures of agents in the environment, each agent periodically generates *AgentAlive* heart-beat events. The *AgentAlive* detector in an agent is triggered by a local *TimerEvent*. The agent responsible for detecting the failures of another agent subscribes to these heart-beat events from that agent. For this purpose it contains the *AgentFailure* detector. Each failure detector agent

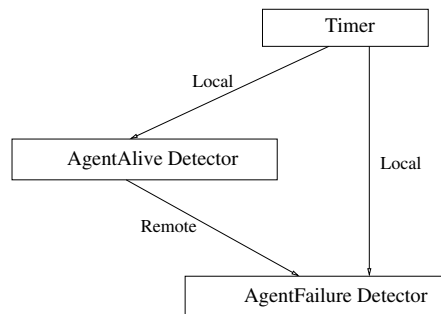


Figure 5. Trigger dependency—example 2.

may also itself generate *AgentAlive* events, to report its own status to some other agent responsible for detecting its failures. We require that the *AgentFailure* detector in an agent should only be triggered by remotely generated *AgentAlive* events and not by the local events, as doing so is useless. This is specified in the definition of the *AgentFailure* detector that it should only be triggered by the local *TimerEvents* and remotely generated *AgentAlive* events. This is shown in Figure 5.

3.2. System management agent

In our original design the SMA's functionality included the creation and configuration of agents according to a system-level configuration specified by the administrators. The configuration of the monitoring system was specified through a set of *configuration files*. The configuration files completely specified the detectors/handlers to be installed in each of the agents. Moreover, they also specified the event subscription/notification relationships for each of the agents. To install a given configuration, the SMA consulted the configuration files and created agents with the appropriate detectors/handlers. It then dispatched agents to the appropriate hosts. The system administrator had to create the configuration files according to the system-wide monitoring goals. The SMA maintained a configuration database to store the configuration of each agent present in the system. This database information was used to restart an agent after its failure.

There were three main drawbacks of the design discussed above. First, the SMA became a bottleneck when large number of agents had to be deployed in the domain. Second, the SMA's knowledge about the detectors/handlers deployed in an agent and the agent's event subscription/notification relationship were maintained in a set of configuration files. These files had to be changed for performing any modification to the network monitoring configuration, such as adding new agents in the domain, adding new detectors/handlers to an agent, or changing the event subscription/notification relationship between agents. Third, the SMA also had to checkpoint the configuration of each agent for failure recovery.

The policy-based design of the monitoring system overcomes these drawbacks as we show in the following sections. In the policy-based design, the SMA is no longer required to launch agents. The agent server on each host creates and starts the monitoring agent on its host. The agent then informs the SMA about its arrival. Second, the SMA is given the system-level policies in the form of

event/action rules. These policies are triggered when the SMA receives notification of agent arrivals or failures. The policy handling action of the SMA consists of configuring the agents with appropriate detectors and handlers as dictated by the policy rules. The SMA also provides each agent with policies for establishing event subscription/notification relationships with other agents in the domain. The functionality of the SMA is kept at a minimal level in this policy-based approach. Also, the system becomes scalable, as agents are no longer launched from the SMA.

4. SYSTEM MANAGEMENT AGENT DESIGN FOR POLICY-BASED AUTONOMIC MANAGEMENT

An agent-based system's functional and non-functional requirements identify the high-level goals that should be fulfilled by the system. These goals are used to define the *system level policies* of the agent environment. Policies are represented as rules that specify the execution of certain actions when the events (called *policy events*) affecting the policies occur. These policies include functional requirements, such as which agents or components should be installed for the required functionality, how many instances of various agents are required, how these agents should interact with each other, and what type of agent should be installed on a host. These policies also include non-functional requirements such as failure monitoring and timely recovery of agents in the system.

In order to satisfy the *system level policies*, components with various functionalities may need to be installed on agents in the domain. The *component integration policies* identify the constraints and requirements for installing a component within an agent. A component when installed within an agent may require co-location of other components in that agent. Components may depend on events generated by components located in remote agents. An agent needs to interact with other agents in order to satisfy the event subscription/notification requirements of its components.

In this approach, it is important to monitor policy violations and perform policy enforcement actions. As new agents and components are deployed in such an environment, they need to be configured according to the system-level policies. Similarly, when an agent/component is removed from the system, due to failures or administrative reasons, appropriate reconfiguration actions are needed to be executed to preserve the invariants implied by the policies. Therefore, arrivals and departures of agents need to be monitored. System level services and mechanisms are required to monitor policy events and execute the appropriate policy enforcement actions, which may involve execution of configuration control functions.

In the policy-based design of the network monitoring system, the SMA performs the enforcement of system level policies. The system level policies determine an agent's functionality and interaction relationships with the existing agents. High-level network monitoring goals are translated into policies and given to the SMA in the form of event handlers for various kinds of policy events. The SMA is responsible for evaluating policies and enforcing them when critical changes occur in the system state. Policy enforcement actions may require the SMA to install/remove components on agents in the system. The SMA maintains a configuration database for obtaining information about agents. Various agent-related queries can be performed on this database. Examples of such queries are, finding all agents in the system having a particular attribute, or finding an agent having certain specific set of attributes.

Agent servers report the creation or arrival of new agents to the SMA. The policy enforcement action corresponding to the *AgentArrival* events consists of deploying appropriate detectors and

handlers according to these policies. Each agent is also deployed with the appropriate event subscription/notification policies for establishing event relationships with other agents in the system. The policy enforcement action may also require the SMA to inform some of the existing agents about the arrival of a new agent, thus facilitating event subscription from the newly deployed agent.

Another type of policy event in the system corresponds to the failure of an agent or its components. The mechanisms for failure detection are discussed in the next section. To restart a failed agent, the SMA creates, configures, and relaunches the agent to an appropriate host, according to the system level policies. The autonomic mechanisms for restoring a restarted agent's event subscription/notification relationships are detailed in the following section. In our approach, an important design principle is to require each restarted agent to perform its own state recovery, so that the SMA is only minimally involved in maintaining any information about an agent's state. Similarly, components within an agent should be designed to reconstruct their execution state, through checkpoints or other mechanisms such as soft-state [11,12].

In the following sections we present examples of some of the policies used in the network monitoring system that are derived from the high-level monitoring goals. In general, the policies derived from a given set of high-level requirements reflect a specific design approach to achieve those requirements.

4.1. Requirement 1

One of the robustness requirements of the network monitoring system is that all the agents in the system should be monitored for failures. Each agent should contain a heart-beat monitor component and should send *AgentAlive* events to an agent containing the *AgentFailure* detector. The following policies are derived from the above requirement. These policies reflect a specific design approach in which a set of agents is responsible for detecting the failures of all other agents. Some other design approach would result in a different set of policies.

1. All agents should be deployed with the *AgentAlive* detector.
2. There should be at least two agents deployed with the *AgentFailure* detector. These agents will act as failure detector agents for all other agents in the system and also for one another.
3. All agents should send locally generated *AgentAlive* events to the failure detector agents.

The SMA enforces the above policies by performing the following actions. On receiving an agent-arrival notification, the SMA installs the *AgentAlive* detector on the agent. The first two agents arriving in the system are designated as the failure detector agents (FDAs) for all other agents. FDAs establish event subscriptions with all other agents for *AgentAlive* events. If one of the FDA fails, the SMA designates another agent for performing the failure detection function. The new FDA subscribes *AgentAlive* events from all the agents present in the system.

4.2. Requirement 2

One of the network monitoring goals is to detect abnormally high numbers of root-login activities (successful/unsuccessful attempts) in the entire domain in a specified interval of time. This goal is translated into the following policies.

1. All agents in the system should contain the *Login detector*.

2. All the components required to detect the *RootLogin* event should be installed on the agent monitoring a host. These components correspond to the detectors presented in Figure 4.
3. At least one agent should be deployed with the *AbnormalRootLogin* detector in the system. This is the root login correlator agent.
4. All the agents should send locally detected root login events to the root login correlator agent.

The SMA enforces the above policies by performing the following actions. On receiving an agent-arrival notification, the SMA installs on the agent all the necessary detectors for root login detection. It also designates the first arriving agent as the root login correlator agent. This root login correlator agent subscribes *RootLogin* events from each subsequently arriving agent. If the root login correlator agent fails, the SMA designates another existing agent as the root login correlator agent. The new root login correlator agent then subscribes to *RootLogin* events from all the agents in the system.

4.3. Requirement 3

One of the goals of the network monitoring system is to continuously monitor the local network traffic involving the protocols and the ports listed in the alerts posted in the known attack vectors on the CERT Web site [13]. The following policies are derived from this requirement.

1. There should be a monitoring agent in the system that continuously monitors the CERT Web site to find new alerts listed on the site. This agent generates *CERTAlert* events corresponding to any new (protocol, port) appearing in any new alerts listed on the CERT Web site.
2. There should be at least one agent monitoring the network traffic involving vulnerable port numbers appearing in the alerts posted on the CERT Web site. This agent should subscribe to the *CERTAlert* events generated by the agent monitoring the CERT Web site. On receiving such an event, it should appropriately update its rules for monitoring the network traffic. On detecting such traffic, it should generate an appropriate alarm for the system administrators.

The SMA enforces the first policy by installing on any one of the agents in the system a detector that periodically monitors the CERT Web site. If this agent fails then one of the other agents is deployed with this detector.

The second policy is enforced by creating and launching an agent that uses the Snort [14] utility to monitor the network traffic with rules that check for traffic using any of the vulnerable (protocol, port) pairs listed on the CERT Web site. There is an administrative constraint in our domain, which requires that any agent dealing with network traffic should run only on the host that also runs Snort [14], a packet filtering application. This constraint dictates the failure handling policy for this agent—it can be restarted only on the hosts that run Snort.

4.4. XML-based policy specification framework

We have developed and implemented an XML-based policy specification framework for describing system level policies that are enforced by the SMA. The design of this XML policy specification framework has been driven by the requirements for performing policy-based autonomic management, and is based on the Event–Condition–Action (ECA) model [15]. It closely resembles policy languages such as Ponder [16] and PDL [17].

```

1. <POLICY>
2.   <ONEVENT>
3.     <EVENT NAME="Event1"/>
4.     <EVENT NAME="Event2"/>
5.     <POLICY-ACTION>
6.       <DEFINITION OBJECT="Object-Name"> ...
7.     </DEFINITION>
8.     <RULE>
9.       <CONDITION> ...
10.      </CONDITION>
11.      <ACTION OBJECT="Object-Name">
12.        <OPERATION NAME="Operation-Name" ... />
13.      </ACTION>
14.    </RULE>
15.  </POLICY-ACTION>
16. </ONEVENT>
17. </POLICY>

```

Figure 6. Policy example template.

The most important functional requirement of a policy-based system is to be able to monitor policy violations, and perform policy enforcement actions if any policies are violated. A set of policy-events need to be identified, the occurrence of which may signify policy violations. Policy enforcement actions consist of performing certain actions on a set of components satisfying particular criteria.

4.4.1. Policy specification

The policy definition construct is based on the ECA model. It defines a set of events that would trigger the evaluation of the condition that determines if a set of policy enforcement actions are needed to be performed. The condition–action part is called the rule of the policy. As shown in Figure 6, a policy definition is enclosed within the POLICY tags. A set of such policy definitions may be specified for a system. Within the ONEVENT tags, one or more trigger events are defined with the EVENT tag and the RULE part of the policy is defined within the POLICY–ACTION tag. The POLICY–ACTION may also contain definitions of sets of agents selected from the configuration database, satisfying certain properties. A set of agents is defined and selected using the constructs supported by the DEFINITION and SELECT tags. The condition and action part of the rule can perform operations on such a set of agents. The RULE part of a policy contains the CONDITION and ACTION tags defining the condition–action pair, as detailed below. In the configuration database, a table named AGENT-SET holds information about all the agents present in the domain at any given time.

4.4.2. Condition specification

A boolean condition can be defined for the state of an agent or a set of agents through the CONDITION tag. With the condition construct one can test the attribute of an object or the properties of a set,

```

1.      <DEFINITION SET="FDA-SET">
2.          <SELECT FROM="AGENT-SET" QUALIFIER="ALL" OBJECT="X">
3.              <CONDITION>
4.                  <EQUAL>
5.                      <ATTRIBUTE OBJECT="X" NAME="ROLE"/>
6.                      <STRING VALUE="FDA"/>
7.                  </EQUAL>
8.              </CONDITION>
9.          </SELECT>
10.     </DEFINITION>

```

Figure 7. FDA set definition.

such as set cardinality and set membership or subset relationships. A condition definition is used as a part of the **RULE** construct to specify the condition for an action execution, and in the **SELECT** construct for selecting a set of agents from the configuration database satisfying the given condition. An example of the condition construct with **SELECT** is shown in Figure 7 (lines 3–8). This condition evaluates to true if the attribute *ROLE* of the object *X* has the value equal to *FDA*.

4.4.3. Object definition

An object can be defined to represent a set of components selected from the configuration database, satisfying certain criteria. A unique name can be defined to refer to such an object through the **DEFINITION** tag. A **SELECT** tag can be used to perform selection operations on the configuration database. A **SELECT** tag has three attributes: (1) attribute *FROM* to specify the set from which selection should be made; (2) attribute *QUALIFIER* to specify the number of components that should be selected; and (3) attribute *OBJECT* to refer to each individual component of the configuration database. A **SELECT** tag may optionally enclose a **CONDITION** tag specifying the condition that needs to be satisfied for the component to be selected. For example, in Figure 7 a set *FDA-SET* is defined to refer to a set created by selecting *all* the agents from the set *AGENT-SET* that have the *ROLE* attribute value equal to *FDA*.

4.4.4. Policy enforcement action specification

Policy enforcement is specified through the **RULE** tag in the specification framework. A **RULE** specifies two things: (1) a **CONDITION** tag specifying the system state denoting a policy violation condition; and (2) a set of policy enforcement actions specified through a list of **ACTION** tags. These actions are performed only when the condition is true. A condition can be empty in which case it is considered as being *true*.

An **ACTION** tag may specify the name of an *OBJECT* on which one or more operations are performed. Each operation is specified through the **OPERATION** tag. Multiple such operations may be enclosed in a single **ACTION** tag. Parameters for each operation may be different and such parameters are specified through the attributes of the **OPERATION** tag. If the object specified in the **ACTION** tag

represents a set of agents or components, the operations of the action are performed on each member of the set. If no object is specified, the operation name represents a system-defined function.

We will now show three examples of actions to demonstrate different aspects of the ACTION construct.

- An action consisting of the installation of an *AgentAliveEvent* detector and handler on an object named *NEW-AGENT* can be written as follows:

```
<ACTION OBJECT="NEW-AGENT">
  <OPERATION NAME="INSTALL" DETECTOR="AgentAliveEventDetector"
    HANDLER="AgentAliveEventHandler"/>
</ACTION>
```

- An action consisting of setting up event subscription between the objects of the set named *FDA-SET* and an agent named *NEW-AGENT* is shown below. The set of operation attributes specified for this example are different to those used in the first example:

```
<ACTION OBJECT="FDA-SET">
  <OPERATION NAME="SUBSCRIBE" EVENT="AgentAliveEvent"
    FROM="NEW-AGENT"/>
</ACTION>
```

- An action that does not specify any object is shown below. This action consists of a system-defined operation for generating an event of a particular type, in this case *FDAFailureEvent*:

```
<ACTION>
  <OPERATION NAME="GENERATE-EVENT" EVENT="FDAFailureEvent" />
</ACTION>
```

4.4.5. Example policy specification

In Figures 7–10 we present the policy specification for Requirement 1, as stated in Section 4.1. Figure 7 presents the definition of the set *FDA-SET*. This set is formed by selecting all the agents in the domain for which the *ROLE* attribute value is equal to *FDA*. This set is referenced by all the three policies in Figures 8–10. Hence the definition of the set is specified as a global definition. It is evaluated in the context of the policy in which it is included.

Figure 8 denotes a policy that is triggered by *NewAgentArrivalEvent* corresponding to the arrival of a new agent in the domain. The purpose of this policy is to configure the arriving agent correctly according to the high level requirements. An object with the name *NEW-AGENT* is defined to refer to the arriving agent (line 6). The value of the attribute *URN* for the *NewAgentArrivalEvent* is assigned to this object (lines 7–9). The policy enforcement corresponding to this event consists of two actions specified through the *RULE* construct (lines 11–20). The first action consists of installing an *AgentAliveEvent* detector and handler on the newly arrived agent (lines 13–15). The second action

```

1. <POLICY>
2. <ONEVENT>
3.   <EVENT NAME="NewAgentArrivalEvent"/>
4.   <INCLUDE-DEFINITION NAME="FDA-SET" />
5.   <POLICY-ACTION>
6.     <DEFINITION OBJECT="NEW-AGENT">
7.       <ACTION OBJECT="NewAgentArrivalEvent" >
8.         <OPERATION NAME="GET-ATTRIBUTE" ATTRIBUTE="URN"/>
9.       </ACTION>
10.    </DEFINITION>
11.    <RULE>
12.      <CONDITION/>
13.      <ACTION OBJECT="NEW-AGENT">
14.        <OPERATION NAME="INSTALL" DETECTOR="AgentAliveEventDetector"
15.          HANDLER="AgentAliveEventHandler"/>
16.      </ACTION>
17.      <ACTION SET ="FDA-SET"
18.        <OPERATION NAME="SUBSCRIBE" EVENT="AgentAliveEvent"
19.          FROM="NEW-AGENT"/>
20.      </ACTION>
21.    </RULE>
22.  </POLICY-ACTION>
23. </ONEVENT>
24. </POLICY>

```

Figure 8. Agent arrival policy.

consists of setting up event subscription/notification for *AgentAliveEvent* between this agent and the agents in the *FDA-SET* (lines 16–19).

Figure 9 denotes a policy that is triggered by *AgentFailureEvent*. The purpose of this policy is to filter the *AgentFailureEvent* to determine if it corresponds to the failure of a FDA agent. Correspondingly, the object *FAILED-AGENT* is defined that refers to the URN of the failed agent (lines 5–9). The policy evaluation rule specifies the condition (lines 11–13), which evaluates to true if the *FAILED-AGENT* is a member of the set *FDA-SET*. The action specified for this policy consists of generating the *FDAFailureEvent* (line 15).

The purpose of the policy specified in Figure 10 is to configure the failure detector agents in the domain. It is triggered by two events: *NewAgentArrivalEvent* (line 3) and *FDAFailureEvent* (line 4). The DEFINITION construct (lines 6–15) creates an object with the name *SOME-AGENT* that refers to any agent from the agent database for which the *ROLE* attribute value is *not equal* to *FDA*. The RULE construct (lines 16–28) specifies the policy violation condition (lines 17–22) and the policy enforcement actions (lines 23–27). The boolean condition checks if the cardinality of the (*FDA-SET*) is less than two.

Three operations are specified to be performed on the agent identified with *SOME-AGENT*. The first operation consists of installing *FailureEventDetector* and *FailureEventHandler* on this agent (line 24). The second operation consists of setting the value of the attribute *ROLE* to *FDA* for this agent (line 25).

```

1. <POLICY>
2.   <ONEVENT>
3.     <EVENT NAME="AgentFailureEvent"/>
4.     <POLICY-ACTION>
5.       <DEFINITION OBJECT="FAILED-AGENT">
6.         <ACTION OBJECT="AgentFailureEvent" >
7.           <OPERATION NAME="GET-ATTRIBUTE" ATTRIBUTE="URN"/>
8.         </ACTION>
9.       </DEFINITION>
10.      <RULE>
11.        <CONDITION>
12.          <ISMEMBER SOURCE="FAILED-AGENT" TARGET="FDA-SET"/>
13.        </CONDITION>
14.        <ACTION>
15.          <OPERATION NAME="GENERATE-EVENT" EVENT="FDAFailureEvent" />
16.        </ACTION>
17.      </RULE>
18.    </POLICY-ACTION>
19.  </ONEVENT>
20. </POLICY>

```

Figure 9. FDA failure event generation policy.

The third operation consists of setting the subscription for *AgentAliveEvent* between this agent and all the other agents present in the system (line 26).

For the first two agents arriving in the domain, the condition specified in lines 17–22 is satisfied. This will trigger the set of operations specified in lines 23–27. This will cause the newly arrived agent, identified by *SOME-AGENT*, to become the FDA. For all the subsequent agents this condition will not be satisfied as there will be two FDAs already present in the domain.

This policy is also triggered by the *FDAFailureEvent* that is generated when one of the FDAs fails. This will cause the condition specified in lines 17–22 to become true. Some other agent will be selected to become the FDA. The policy enforcement action in line 26 will ensure that this new FDA will get *AgentAlive* events from all the agents in the domain, including the other FDA. Multiple policies that are triggered by the same event are processed in the order in which they are deployed in the domain. In the network monitoring system the policy in Figure 8 is deployed before the policy in Figure 10. This causes policy in Figure 8 to be evaluated first on the receipt of *NewAgentArrivalEvent*.

5. AUTONOMIC CONFIGURATION AND RECOVERY OF AGENTS

We describe here the autonomic mechanisms for configuration and recovery of agents. Our design was driven with the goal of performing autonomic recovery and repair of failed agents within a short time-span, typically in the range of a few minutes. The primary objective is to perform the recovery and

```

1. <POLICY>
2. <ONEVENT>
3. <EVENT NAME="NewAgentArrivalEvent"/>
4. <EVENT NAME="FDAFailureEvent"/>
5. <INCLUDE-DEFINITION NAME="FDA-SET"/>
6. <DEFINITION OBJECT="SOME-AGENT">
7. <SELECT FROM="AGENT-SET" QUALIFIER="ANY" OBJECT="X">
8. <CONDITION >
9. <NOTEQUAL>
10. <ATTRIBUTE OBJECT="X" NAME="ROLE"/>
11. <STRING VALUE="FDA"/>
12. </NOTEQUAL>
13. </CONDITION>
14. </SELECT>
15. </DEFINITION>
16. <RULE>
17. <CONDITION>
18. <LESS-THAN>
19. <ATTRIBUTE OBJECT="FDA-SET" NAME="SIZE"/>
20. <INTEGER VALUE="2"/>
21. </LESS-THAN>
22. </CONDITION>
23. <ACTION OBJECT="SOME-AGENT">
24. <OPERATION NAME="INSTALL" DETECTOR="FailureEventDetector"
    HANDLER="FailureEventHandler"/>
25. <OPERATION NAME="SET-ATTRIBUTE" ATTRIBUTE="ROLE"
    VALUE="FDA"/>
26. <OPERATION NAME="SUBSCRIBE" EVENT="AgentAliveEvent"
    FROM="AGENT-SET"/>
27. </ACTION>
28. </RULE>
29. </POLICY-ACTION>
30. </ONEVENT>
31. </POLICY>

```

Figure 10. FDA regeneration policy.

repair of failed agents rather than diagnose the failures. This is to ensure continued monitoring of the infrastructure resources by minimizing human intervention in repairing failures.

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 sections are used as the basic building blocks for failure detection. The publish–subscribe model is used for notifying failure conditions to other agents that need to participate in recovery and reconfiguration. Our design relies on the periodic monitoring of an agent for failure conditions and the continuous generation of the failure events until either the

failed agent is repaired or the system administrator performs explicit configuration changes to bypass the failure.

5.1. Failure modes and recovery requirements

The various failure modes in our system include host failures, agent server failures, agent failures, and detector failures within an agent. Agent servers or hosts may crash bringing down the entire set of monitoring mechanisms at that host.

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 agents. 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 in an agent can be remotely un-installed and replaced with a new detector by an agent performing recovery actions. It is also possible to remotely terminate a failed agent and create a new one in its place.

The state of an agent consists of various detectors and its event communication relationships with other agents in the system. With the policy-based approach, the SMA does not maintain any checkpointed configuration state of any agent for recovery. The goal of the policy-based techniques is to appropriately configure an agent on its restart. Moreover, an agent is responsible for checkpointing its state that needs to be preserved across a failure and restart. In the network monitoring system, most of the detectors are either state-less or they maintain *soft state* [11,12], which can be reconstructed through interactions with other agents during recovery. Information about any new subscriptions registered by an agent during its execution is not checkpointed. This also forms the soft state, which is recreated on restart, as detailed below. The motivation for this is to keep the checkpointing overhead low for the agents.

Many of the detectors in our system generate events by periodically monitoring the system log-files for new events recorded in them by the host operating system. The soft state of a detector monitoring a log-file consists of the position offset in the log-file for the last processed event. To restore this state on restart, the detector needs to determine the most recent event that it processed from the log-file. One approach to determine this information is by querying the event subscribers. The other approach is that the detector explicitly stores its current position offset in a local checkpoint file.

5.2. Agent-level self-configuration

The self-configuration mechanism of an agent involves integration of detectors/handlers within an agent, and registering event subscriptions at remote agents.

5.2.1. Agent level self-monitoring

Agent-level self-monitoring is required to ensure that various components deployed on the agent are functioning properly. Each agent is equipped with an *AgentAlive* 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 three items: (1) a list of detectors that are considered to be correctly functioning in the agent; (2) the list of its current subscriber agents and the events that they are subscribing to; and (3) a *current configuration number*. Whenever an agent's configuration

is changed with the addition/deletion of a detector or a subscriber, this number is incremented. This number is also incremented when an agent is re-launched on recovery. The purpose of the configuration number is two-fold: first, it makes sure that the subscribers of an *AgentAlive* message become aware of any configuration changes, and the failure detector agents are able to ignore any *AgentAlive* events related to old configurations.

5.2.2. Detector self-configuration

Detector configuration policies identify how the detectors co-located within an agent should be configured. Detectors are required to be designed to be self-configurable within an agent, and each agent provides a generic framework for self-configuration of its components when created or restarted. Each detector's code specifies the names of its default-triggering events. When a new event detector is installed in an agent, the configuration dependencies of that detector are resolved by the agent by establishing event subscription relationships with other agents in the system.

5.2.3. Self-configuration of event subscriptions

When an agent is restarted at a host, it does not have any information about its subscriber agents. The subscribers are required to register themselves with the event publishers to get the events of interest. Therefore, when an agent failure is detected, it is required that all of its subscribers are notified 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 FDA sends the *AgentFailure* event to all the subscribers of the failed agent. On receiving this *AgentFailure* event, a subscriber agent puts the failed agent's name and the list of subscribed events from that agent in the *Outstanding Subscription List*. Its *Subscription Manager* thread then periodically attempts to re-register these event subscriptions with the failed agent, until successful.

5.3. Peer-to-peer cooperative failure detection and recovery by agents

Any agent in the system can perform the function of detecting the failure of another agent if it has an *AgentFailure* detector object. It only needs to subscribe to the *AgentAlive* events from the agents it wants to monitor. Similarly, any agent can be entrusted with the task of performing the recovery of a failed agent if it has the appropriate handler object for the corresponding *AgentFailure* event.

The function of the *AgentFailure* detector is to detect an agent failure condition and generate an *AgentFailure* event. This event indicates either a *crash-failure* of an agent (e.g. due to a host crash) or a *partial failure*, indicating failures of a subset of the components within an agent. The execution of the failure detector is triggered either by a local timer or by a remote *AgentAlive* event, as shown in Figure 5. An agent containing the *AgentAlive* detector periodically generates the *AgentAlive* events. These are sent to one or more agents responsible for detecting this agent's failure.

An *AgentFailure* detector receives these periodic *AgentAlive* events. It also has configuration information for each agent, and its function is to process the *AgentAlive* events and generate failure events. The absence of a certain number of consecutive *AgentAlive* events from an agent indicates a *crash-failure*. On the other hand, failure of a detector within an agent is deduced by comparing the list of currently present detectors reported in an *AgentAlive* event with the reference configuration for

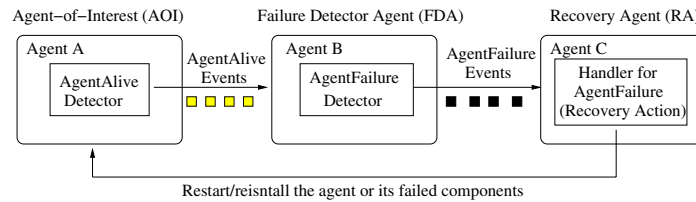


Figure 11. Model for peer-based cooperative recovery.

that agent, which is created whenever the configuration number in the *AgentAlive* event from an agent increases. Based on any mismatch detection, a *partial-failure* event is generated containing a list of the failed components.

An example pattern using the basic model for peer-to-peer failure detection and recovery is shown in Figure 11. Here agent *A* is our *agent-of-interest* (AOI), which is to be made resilient. Subject to the security constraints and other policies, either the configuration service or agent *A* itself assigns the task of detecting *A*'s failure to agent *B* and the recovery task to agent *C*. Here *B* serves as the FDA and *C* functions as the *recovery agent* (RA) for agent *A*. For implementing these interactions, an *AgentAlive* detector is installed in *A*, an *AgentFailure* detector is installed in *B*, and a handler for *AgentFailure* events is installed in *C*. Agent *B* generates *AgentFailure* events if it stops receiving any *AgentAlive* events from *A*, or if the reported set of functioning detectors in the *AgentAlive* events deviates from the agent's reference configuration. The *AgentFailure* events are continuously generated at some fixed interval until an *AgentAlive* event from *A* is once again received by the failure detector in *B*. These failure events are being received by agent *C*, in which a handler for these types of events is installed. This handler performs recovery of *A* by either restarting the agent or re-installing any of its failed detectors. Here it should be noted that the failure detectors are persistent in continuously reporting failure events until recovery is completed. This implies that the handler performing the recovery action needs to be designed to perform the recovery action only once, even when it continues to receive the failure events during the execution of the recovery action. It should also be noted that this peer-based cooperative recovery is programmed using the basic event-handling model presented in Section 2.2. No new or additional mechanisms are required.

There can be multiple agents performing the failure detection and recovery functions for an agent. The above example can be further enriched to make the failure detector and recovery functions resilient by implementing them using a group of agents, as shown in Figure 12. In such a configuration, no coordination is needed among the FDAs. All of them independently detect and report any failure event. However, the recovery agents need to coordinate their actions if an *AgentFailure* event is reported. If the recovery actions are not idempotent, then only one of the recovery agents should perform these actions. For example, the agents in a group may follow the protocol that the agent with the smallest (or the largest) id executes the recovery actions. The agents in a group can maintain the current list of group members using the Konark event model and using *AgentAlive* events. Different patterns of interactions between agents and their failure detectors can be specified as system wide policies. The handling of other forms of failures, such as host failures and agent server failures, requires manual intervention. Agent servers can be remotely started on a host if the SSH daemon is running on it.

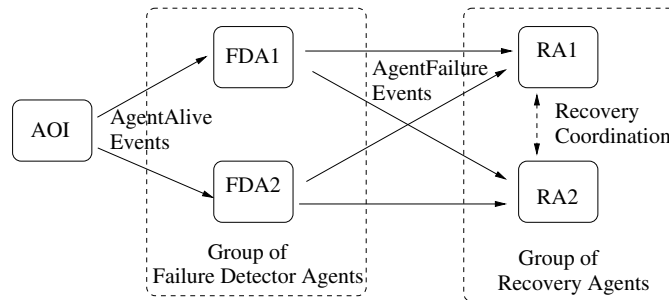


Figure 12. Cooperative agent groups for failure detection and recovery.

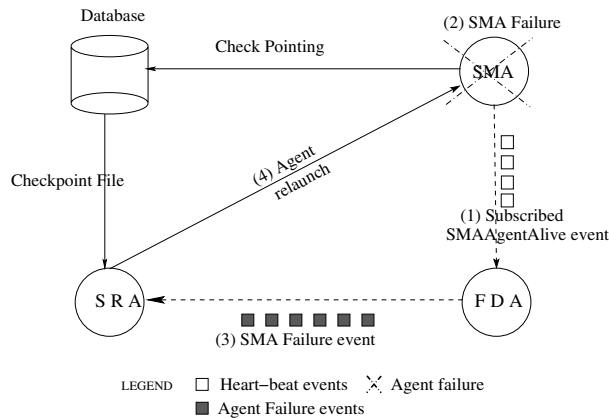


Figure 13. Failure recovery of the SMA.

5.4. Recovery of system management agent

The robustness of the SMA can be achieved by either replicating it, or periodically checkpointing the SMA and using another agent to recover the failed SMA. In our design we used the second approach as the state of the SMA gets modified infrequently. Moreover, we wanted to use the same mechanisms to recover the SMA as used for the failure recovery of the other agents in the system. Figure 13 shows the recovery of the SMA. One of the agents is assigned the function of recovering a failed SMA. This agent is called the *SMA Recovery Agent* (SRA). Both the SMA and the SRA subscribe to each other's *AgentFailure* events from the FDAs. These two agents execute on different hosts. Any agent can perform the task of the SRA by appropriately installing the required handler to perform SMA recovery when it receives an *AgentFailure* event for the SMA. This handler creates the SMA using its

most recent checkpoint file and launches it to another host for execution. To recover the failed SRA, the SMA restarts it exactly as in case of any other agent.

6. EXPERIMENTAL EVALUATION OF POLICY-BASED MECHANISMS FOR AUTONOMIC CONFIGURATION AND RECOVERY

In this section we evaluate the policy-based mechanisms of Konark framework along two dimensions. First, we evaluate the policy-based design of the network event monitoring application. Second, we evaluate the performance of Konark's mechanisms for autonomic configuration and failure recovery.

In all our experiments we use Intel i686 machines running Linux 2.4.27-5-686-smp for deploying SMAs and FDAs. Monitoring agents are run on 10 SUN sparc machines (sun4u sparc SUNW, Sun-Blade-100/Sun-Blade-1500) running SunOS 5.8. Every experiment, corresponding to any given agent configuration, was run up to 10 times to ensure that the 90% confidence interval was within 10% of the reported average values.

6.1. Evaluation of policy-based network monitoring system

The goal of this evaluation was two fold: first, we wanted to evaluate the practicality of the policy-based design for agent-based network monitoring in a real world environment. Second, we wanted to evaluate the performance of the policy-based approach in comparison to our non-policy-based original approach. Specifically, the policy-based system could become a practical design option only if it performed at least as good as the non-policy-based design. With these objectives, we performed the experiment of deploying the complete configuration in our testbed environment using both policy-based and non-policy-based approach.

The network event monitoring system is used for monitoring 20 hosts in our Department's Graduate Lab environment. These machines are monitored for the following critical events: runaway processes, processes running with root privileges, abnormal root logins from outside the domains, multiple root logins within a specified interval on any host, multiple root logins within a specified time on more than one host, and correctly running daemon services. Detectors and handlers corresponding to these monitoring requirements are deployed on agents running on each host.

In our initial design, the SMA was provided with this information through a set of configuration files. It created, configured, and launched agents to all the hosts. For the robustness requirement, each agent was also deployed with an *AgentAlive* detector. One agent was deployed with the *AgentFailure* detector and was also configured to receive *AgentAlive* events from other agents. This agent sent the *AgentFailure* events to the SMA, who in turn launched the agents to the appropriate hosts. Observed times for configuring the entire system, consisting of 20 agents each deployed with 10 detectors, were in the range of two to three minutes.

In the policy-based design, the high-level network monitoring goals remain the same. The policy-based design differs from the earlier design in the way agents are configured and launched in the environment. In this design, agent servers on each host start the agents and notify their arrival to the SMA. These agent arrival notifications are the policy events in the system. The SMA handles the policy events by deploying detectors and handlers corresponding to the previously mentioned monitoring requirements. Thus in the policy-based design, the participation of the SMA is kept to a minimal level.

The average time for configuring the entire system, consisting of 20 agents each deployed with 10 detectors, in the policy-based approach was 29.38 seconds (standard deviation 0.99). This improvement in the system configuration time for the policy-based design occurs because the SMA is no longer required to configure and launch agents in the environment. Extending the configuration of the system is also straightforward in the policy-based approach. Any new agent that needs to be deployed in the system needs to know only the URN of the SMA. Once the agent contacts the SMA, it is configured correctly according to the system-wide policies. Thus no modifications are required in any of the configuration files making the task of adding a new agent extremely straightforward.

6.2. Performance evaluation of autonomic mechanisms for configuration and recovery

We evaluate the performance and scalability of the mechanisms for autonomic configuration and failure recovery by seeking answers to the following questions related to Konark's performance.

1. *What is the cost of configuring large number of agents?* The configuration cost is a key piece of information that is useful in making configuration design decisions for large agent systems. With this information, a system administrator can design different clustering strategies to satisfy the configuration constraints corresponding to the system monitoring goals. We consider the case of *flash-arrival* of agents in the experiments. This corresponds to large number of agents arriving within a short period.
2. *What is the performance of autonomic mechanisms for failure detection and recovery?* Failure of an agent/component can lead to a potential policy violation. Failure detection and recovery cost provides an indication of the delay in taking corrective actions corresponding to the potential policy violations due to the failure. We consider the case of *flash-crash* of agents in the experiments. This corresponds to a large number of agents failing together.

6.3. Autonomic configuration

We ran a single SMA in the domain and varied the arriving agent population from 200 to 1000 agents by varying the number of agents started on each machine between 20 and 100 for each successive experiment. Two FDAs were run at a known location in the system. Each injected agent was provided with the location information of the SMA. Each agent reported its arrival to the SMA. On receiving arrival notification from an agent, the SMA deployed an *AgentAlive* detector and handler on that agent. Each agent was configured to generate *AgentAlive* events at a period of 10 seconds and send these events to the FDAs. Each unique *AgentAlive* event received by the FDAs was time-stamped for performance measurements. The system configuration time for an agent population of a given size was measured as the difference between the receipt of first agent arrival notification by the SMA to the receipt of the last unique *AgentAlive* event at the FDAs.

Each arriving agent has to look-up the Ajanta Name Registry to translate URNs of the SMA and the FDAs to their respective URLs. Similarly, the SMA has to translate the URN of the incoming agent to its URL in order to deploy components on the agent. The system configuration time is measured as the time from the first agent arrival until the reception by the FDA of the first *AgentAlive* event from the last agent created in the system. This includes the time spent by the arriving agent and the

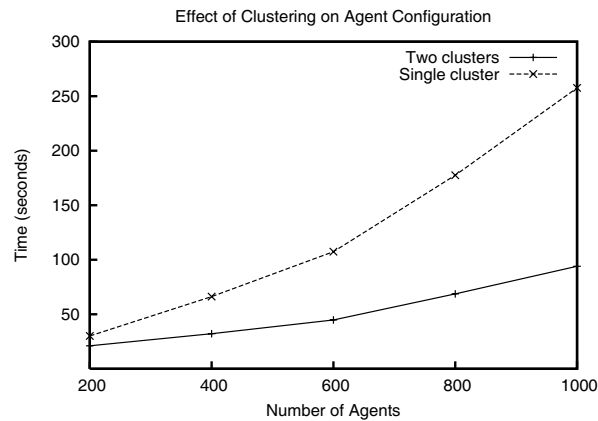


Figure 14. Agent population configuration time.

SMA in the translation process within the Name Registry. Our initial experiments indicated that the Ajanta Name Registry was the performance bottleneck due to the costs imposed by the authenticated communication.

The goal of this experiment was to measure the autonomic configuration performance and aid system administrators in developing suitable clustering strategies for deploying large-scale agent systems. Correspondingly, we compared performance of the system deployed with one and two clusters. All the agents belonging to a single Ajanta namespace are part of one cluster. Two clusters were created by creating two separate Ajanta namespaces managed by two different Ajanta Name Registries. Figure 14 shows the variation of agent population configuration time for different agent populations for the one and two cluster systems.

This agent configuration behavior can be used as a guideline for satisfying the high-level performance goals related to the configuration of large agent systems. For example, consider a system that consists of 600 agents and has a high-level goal that the system configuration time should not exceed 50 seconds. From Figure 14 we see that this constraint can be satisfied by creating two clusters. On the other hand, a single cluster is able to handle only 300 agents within the specified performance constraint.

6.4. Autonomic failure recovery

In the second set of experiments, we evaluated failure recovery characteristics. Each agent was configured to report *AgentAlive* events to the FDAs after every 10 seconds. FDAs generated the first *AgentFailure* event on missing three consecutive *AgentAlive* events from an agent. This *AgentFailure* event was sent to the SMA, which then created and relaunched the agent to the appropriate agent server. After the agent is relaunched it contacted the SMA to indicate its arrival in the domain. The SMA then configured it with the *AgentAlive* detector and handler.

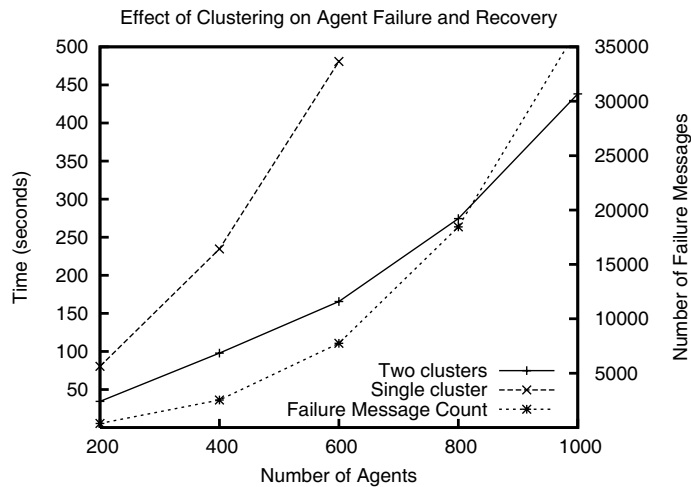


Figure 15. Failure recovery: total time and number of *AgentFailure* events generated.

Failure detection and recovery time is measured as the difference of time between the reception of the first failure event and the first *AgentAlive* event from the last restarted agent *after* the first failure event has occurred. The period of *AgentAlive* event generation and the threshold number of missed *AgentAlive* events leading to *AgentFailure* event generation, can affect the failure detection time. Hence we measure recovery time *after* receiving the first *AgentFailure* event so that it is independent of these factors.

A driver program was used for remotely terminating agents running on an agent server without terminating the agent server itself. This driver program used the remote agent termination API provided by the Ajanta system. Figure 15 plots the failure recovery time for different agent populations configured as one and two cluster systems.

The effect of clustering on the failure recovery performance is clearly seen from these experiments. The Ajanta Name Registry is involved in the following steps of failure recovery process. First, the driver program has to obtain the agent references for killing the agents. Second, the SMA has to obtain the references of the agent server on which the failed agents need to be relaunched. Third, each restarted agent needs to contact the SMA and FDAs according to the agent arrival protocol. When the system was deployed as a single cluster system, the system was able to handle failure recovery of only 600 agents within the specified time limit of 10 minutes. On the other hand, with two clusters the system is able to handle failure recovery of 1000 agents in 7 minutes and 18 seconds.

This agent failure recovery behavior can be used as a guideline for satisfying the high-level performance goals related to the failure recovery of large agent systems. For example, consider a system that consists of 600 agents and has a high-level goal that the failure recovery time should not exceed three minutes. From Figure 15 we see that this constraint can be satisfied only by configuring the system as two cluster system. With two clusters, the system is able to recover 600 agents in 165 seconds.

On the other hand, a single cluster can handle failure recovery of only 200 agents within the specified performance constraint.

Comparing Figures 14 and 15 we see that the failure recovery time for a given agent population is greater than the corresponding agent population configuration time. In our design, the failure recovery time contains two components: the time required for creating and launching the agent by the SMA after receiving failure notification, and the time required for remotely configuring the agent once the agent has been deployed in the system. The agent configuration time is same in both the cases: initial configuration and configuration after failure recovery. Additional time incurred by the failure recovery process corresponds to the time taken by the SMA to remotely launch failed agents.

Each FDA keeps on generating the *AgentFailure* events until it receives an *AgentAlive* event from the restarted agent. Such persistent generation of *AgentFailure* events can lead to lot of processing overhead at the SMA. There is a tradeoff between the continuous generation of *AgentFailure* events and the timely restarting of the failed agent. At one extreme only a single *AgentFailure* event can be generated by the FDA. However, if this event gets lost in communication then the agent will not be restarted at all. Such a tradeoff is a matter of robustness policy within the domain. In our experiments, we implemented the policy of exponentially delaying the generation of successive *AgentFailure* events. Figure 15 plots the number of *AgentFailure* events generated during the failure recovery in the two cluster system. The average number of *AgentFailure* events generated during failure recovery of 1000 agents was 36 668. During the failure recovery process, agents that are launched early start sending *AgentAlive* events to the FDA. For the 1000 agent configuration, the average number of such *AgentAlive* events generated during the failure recovery period was 3525. Thus a total of 40 193 events were generated during the failure recovery of 1000 agents. On the other hand, the average number of *AgentAlive* events generated in the same amount of time for 1000 agent configuration with *no agent failures* was 29 002. Thus there was a 38% increase in the total number of events generated during the failure recovery process.

7. RELATED WORK

Agent-based systems have been proposed as a suitable software engineering paradigm for building large-scale software systems [18]. The application of agent-based designs for building complex control systems are elaborated in [19] with specific concerns for diagnosis and repair functions needed in such systems to ensure robust operation. Management aspects of open multi-agent systems have been addressed through a three-level model of organizational rules, organizational structures, and organizational patterns in [20], and through the *law governed interactions* in [21,22]. Policies used in this paper define rules of agent interactions similar to the *organizational rules* or the *laws*. Similar notions have also been proposed in the context of *normative multi-agent systems* [23,24]. Other researchers have also considered using a policy-based approach for building agent systems [25]. In their approach policies related to roles, agent authorizations, and agent obligations are expressed through the Ponder policy language and are enforced through an agent-based middleware.

Policy-based management of networked systems is a well-established field with considerable work done in the development of policy specification languages [16,17]. The Internet Engineering Task Force (IETF) has defined a common terminology [26] and an information model for network monitoring policies [27]. The XML-based policy framework presented in this paper follows these efforts by expressing the policies as ECA rules. Our policy specification framework is confined to agent-based systems and closely follows the architectural mechanisms provided in the Konark framework.

The policy specification framework is not a general purpose framework as other policy specification languages, such as Ponder [16] or PDL [17]. An advantage of using an XML-based approach is that the policy specification schema can be easily extended with constructs corresponding to evolving requirements.

Konark's dynamic component integration model and event subscription/notification mechanism supports dynamic configuration of peer-to-peer failure monitoring agents, ECA based-policy expression and enforcement, and flexible inter-agent communication. Event-based publish/subscribe systems have matured and are being used as the standard communication mechanism in distributed systems [28–31]. We embed such an event subscription/notification mechanism within a mobile agent paradigm. We use a *soft-state* approach [32] for failure recovery. In this approach, each agent is required to maintain minimal amount of information necessary for recovery. This corresponds to the location of the SMA in the system. Such an approach avoids the complexity of replica maintenance required in replication-based agent fault tolerance [33].

The Konark framework provides capabilities for realizing the vision of autonomic computing systems [1]. An architectural reference model for autonomic computing, consisting of component interfaces for monitoring, testing, addition/modification of policies, negotiation, and binding, is presented in [34]. The embodiment of these concepts into a working prototype system is presented in [35]. They use *goal-driven policies* [36] to enable appropriate autonomic actions by the agents in the system. The high-level functional and non-functional requirements presented in Section 4 are similar to the notion of goals in *goal-driven policies*. We translate these high-level requirements into *action policies* [36] which are enforced by the SMA.

8. CONCLUSIONS

In this paper we have presented an approach for building policy-based autonomic mechanisms for configuration and recovery in agent-based systems. In this approach policies are defined based on the ECA model and they are expressed in XML. The distributed event processing model of the Konark framework is used for policy enforcement by the SMA. Distributed agents are used to monitor the environment for events that may signify potential violation of policies. Such events are communicated to SMA for enforcement actions. We have successfully integrated these autonomic mechanisms in a distributed agent-based network monitoring application built using Konark. Experimental evaluations using this system demonstrate both the feasibility and the scalability of this policy-based approach for building autonomic and robust agent-based systems.

REFERENCES

1. Kephart JO, Chess DM. The vision of autonomic computing. *IEEE Computer* 2003; 41–50.
2. Tripathi A, Ahmed T, Pathak S, Carney M, Dokas P. Paradigms for mobile agent-based active monitoring. *Proceedings of the 2002 IEEE Networks Operations and Management Symposium (NOMS 2002)*, Florence, Italy, 2002. IEEE Communication Society, 2002; 65–78.
3. Tripathi AR, Koka M, Karanth S, Pathak A, Ahmed T. Secure multi-agent coordination in a network monitoring system. *Software Engineering for Large-Scale Multi-Agent Systems 2002 (SELMAS 2002) (Lecturer Notes in Computer Science, vol. 2603)*. Springer: Berlin, 2003; 251–266.

4. Tripathi A, Karnik N, Vora M, Ahmed T, Singh R. Mobile agent programming in Ajanta. *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, TX, 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 190–197.
5. Kiniry J, Zimmerman D. A hands-on look at Java mobile agents. *IEEE Internet Computing* 1997; (July–August):21–30. Available at: <http://computer.org/internet/>.
6. Tripathi A, Koka M, Karanth S, Osipkov I, Talkad H, Ahmed T, Johnson D, Dier S. Robustness and security in a mobile-agent based network monitoring system. *Proceedings of the International Conference on Autonomic Computing (ICAC-04)*, New York, May 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 320–321.
7. Karnik N, Tripathi A. Security in the Ajanta mobile agent system. *Software—Practice and Experience* 2001; **31**(4):301–329.
8. Tripathi A, Kulkarni D, Ahmed T. Policy-driven configuration and management of agent based distributed systems. *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications (Lecture Notes in Computer Science, vol. 3914)*. Springer: Berlin, 2006; 1–16.
9. Tripathi A, Karnik N, Ahmed T, Singh R, Prakash A, Kakani V, Vora M, Pathak M. Design of the Ajanta system for mobile agent programming. *The Journal of Systems and Software* 2002; 123–140.
10. Moats R. RFC 2141: URN Syntax, May 1997.
11. Candea G, Cutler J, Fox A. Improving availability with recursive micro-reboots: A soft-state system case study. *Performance Evaluation Journal* 2004; **56**(1–3).
12. Clark DD. The design philosophy of the DARPA Internet Protocols. *Proceedings of the ACM SIGCOMM*, Stanford, CA, August 1988. ACM Press: New York, 1988; 106–114.
13. CERT. www.us-cert.gov/channels/techalerts.rdf.
14. Roesch M. Snort-lightweight intrusion detection for networks. *Proceedings of the 13th Systems Administration Conference—LISA*, Seattle, WA, November 1999. USENIX Association: Berkeley, CA, 1999.
15. Ceri S, Widom J. Deriving production rules for constraint maintenance. *Proceedings of the 16th International Conference on Very Large Databases (VLDB'90)*, Brisbane, Australia, 1990. Morgan Kaufmann: San Francisco, CA, 1990; 566–577.
16. Damianou N, Dulay N, Lupu E, Sloman M. The Ponder policy specification language. *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, London, U.K., January 2001 (*Lecture Notes in Computer Science, vol. 1995*). Springer, 2001; 18–38.
17. Lobo J, Bhatia R, Naqvi S. A policy description language. *Proceedings of the 16th National Conference on Artificial Intelligence and the 11th Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence (AAAI'99/IAAI'99)*. American Association for Artificial Intelligence: Menlo Park, CA, 1999; 291–298.
18. Jennings NR. An agent-based approach for building complex software systems. *Communications of the ACM* 2001; 35–41.
19. Jennings NR, Bussman S. Agent-based control systems: Why are they suited to engineering complex systems? *IEEE Control Systems Magazine* 2003; **23**(3):61–73.
20. Zambonelli F, Jennings NR, Wooldridge M. Organisational abstractions for the analysis and design of multi-agent systems. *Proceedings of the 1st International Conference on Agent Oriented Software Engineering (AOSE 2000)*, Limerick, Ireland, January 2001.
21. Minsky N, Ungureanu V. Law-governed interaction: A coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2000; **9**(3):273–305.
22. Minsky NH, Murata T. On manageability and robustness of open multi-agent systems. *Proceedings of Computer Security, Dependability and Assurance (Lecture Notes in Computer Science, vol. 2940)*, Lucena C, Garcia A, Romanovsky A, Castro J, Alencar P (eds.). Springer: Berlin, 2004; 189–206.
23. Lopez F, Luck M, d'Inverno M. A normative framework for agent-based systems. *Proceedings of the 1st International Symposium on Normative Multiagent Systems (NorMAS2005)*, Hatfield, U.K. AISB, 2005.
24. Jones A, Sergot M. The characterisation of law and computer systems: The normative systems perspective. *Deontic Logic in Computer Science: Normative System Specification*, Meyer JJCh, Wieringa RJ (eds.). Wiley: Chichester, 1993; 275–307.
25. Corradi A, Dulay N, Montanari R, Stefanelli C. Policy-driven management of agent systems. *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, London, U.K., January 2001 (*Lecture Notes in Computer Science, vol. 1995*). Springer, 2001; 214–229.
26. Westeriner A, Schnizlein J, Strassner J, Scherling M, Quinn B, Herzog S, Huynh A, Carlson M, Perry J, Waldbusser S. Terminology for policy-based management. *Internet Engineering Task Force RFC 3198*, November 2001. The Internet Society, 2001. Available at: <http://www.ietf.org/rfc/rfc3198.txt>.
27. Moore B, Ellesson E, Strassner J, Westerinen A. Policy Core Information Model—Version 1 Specification. *Internet Engineering Task Force RFC 3060*, February 2001. The Internet Society, 2001. Available at: <http://www.ietf.org/rfc/rfc3060.txt>.
28. Eugster PTh, Felber PA, Guerraoui R, Kermarrec A-M. The many faces of publish/subscribe. *ACM Computing Surveys* 2003; **35**(2):114–131.

29. Carzaniga A, Rosenblum DS, Wolf AL. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* 2001; **19**(3):332–383.
30. Rowanhill JC, Varner PE, Knight JC. Efficient hierarchic management for reconfiguration of networked information systems. *International Conference on Dependable Systems and Networks (DSN'04)*, Florence, Italy, 28 June–1 July 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004.
31. Pietzuch PR, Shand B, Bacon J. A framework for event composition in distributed systems. *Proceeding of the 4th ACM/IFIP/USENIX International Conference on Middleware (Middleware'03)*, Rio de Janeiro, Brazil, June 2003 (*Lecture Notes in Computer Science*, vol. 2672). Springer, 2003. 62–82.
32. Fox A, Gribble SD, Chawathe Y, Bewer EA, Gauthier P. Cluster-based scalable network services. *Proceedings of the 8th International Symposium on Operating Systems Principles (SOSP'97)*, St Malo, France, October 1997. ACM SIGOPS, 1997; 78–91.
33. Guessoum Z, Briot JP, Charpentier S, Marin O, Sens P. A fault-tolerant multi-agent framework. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*. ACM Press: New York, 2002; 672–673.
34. White SR, Hanson JE, Whalley I, Chess DM, Kephart JO. An architectural approach to autonomic computing. *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, New York, 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 2–9.
35. Tesauro G, Chess DM, Walsh WE, Das R, Segal A, Whalley I, Kephart JO, White SR. A multi-agent systems approach to autonomic computing. *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*. IEEE Computer Society: Washington, DC, 2004; 464–471.
36. Kephart JO, Walsh WE. An artificial intelligence perspective on autonomic computing policies. *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*. IEEE Computer Society Press: Washington, DC, 2004; 3–12.