

Policy-Driven Configuration and Management of Agent Based Distributed Systems*

Anand Tripathi, Devdatta Kulkarni, and Tanvir Ahmed
Department of Computer Science
University of Minnesota, Minneapolis MN 55455

ABSTRACT

In this paper, we demonstrate a policy based approach for building and managing large scale agent based systems. We identify different classes of policies for agent based component integration. We also identify the system services and mechanisms that are required for policy based integration of components and their management. The utility of this approach is presented through case studies involving two applications making use of the distributed event monitoring system that we have developed.

1. INTRODUCTION

Building and managing large distributed component systems is becoming an increasingly challenging task. Continuous intervention by system administrators is generally limited in large-scale distributed environments. System support is also needed for reconfiguration and reorganization when systems evolve with the addition of new components.

Component based structuring approaches for distributed systems have sufficiently matured and are widely used today [9]. New challenges in using the component technology are concerned with the integration and building of systems using active components such as agents. We demonstrate here that large scale distributed systems can be autonomically configured and managed through policy-driven integration [10] of agent-based components.

Agent based distributed computing models provide an ideal foundation for policy based component integration. Agents are autonomous entities that can encapsulate and enforce local policies. Autonomous agents are capable of learning and adapting to the new or modified global policies that dictate the interactions among distributed agents. Mechanisms of self-configuration, self-monitoring and recovery can be built using such capabilities.

Agents are the basic building-blocks in our approach. They are considered as *first-class components*. An agent is an ac-

* This work was supported by National Science Foundation grants 0087514 and 0411961.

tive object encapsulating other components. It serves as an execution environment with access privileges. Moreover, an agent may be capable of migrating in the network. In this paper we use the term *component* to refer to objects that are contained in an agent; these objects may be active or passive. This agent model is supported through the Ajanta mobile agent framework [3] [13].

In policy based distributed agent system architectures, there are intra-agent policies for component integration, inter-agent policies for global interactions, and policies to ensure system robustness. Based on these policies, rules are derived for dynamic integration of agents and components in the system. Policies identify the conditions under which new agents/components are needed to be installed in the system or some failed agents/components should be restarted. Policies also identify interactions among distributed agents, and rules for integration of components within an agent.

The contextual information, such as availability of other agents and components in the environment, and different services present in the environment, is available in an agent based distributed system. Such contextual information affects various policies in the environment.

In this paper we present a framework for policy identification in distributed agent systems. We demonstrate the utility of this framework through case studies of two distributed agent systems that we have developed. One of these systems is targeted towards network monitoring using mobile agents [15] and the other is targeted towards building secure distributed collaborative applications from their high-level specifications [12, 11]. Both these systems are based on the Ajanta mobile agent programming framework.

Konark[15] is a network monitoring system which monitors hosts in a network for various kinds of events. In this system, mobile agents are launched to monitor each host in the environment. Each agent contains components, termed *detectors*, to detect events in the environment. New detectors can be added to an agent remotely. Examples of events in this system include user logins, program executions on a host, file modifications and network traffic patterns related to intrusions.

Agents can subscribe to events from other agents to detect compound events based on correlation of events from distributed sources. The execution of a detector in an agent can be triggered by events generated in the same agent or other agents in the environment. The list of local or remote events that can trigger a detector form a triggering relation among detectors. It is termed *trigger dependency*. This makes Konark useful for correlating events occurring

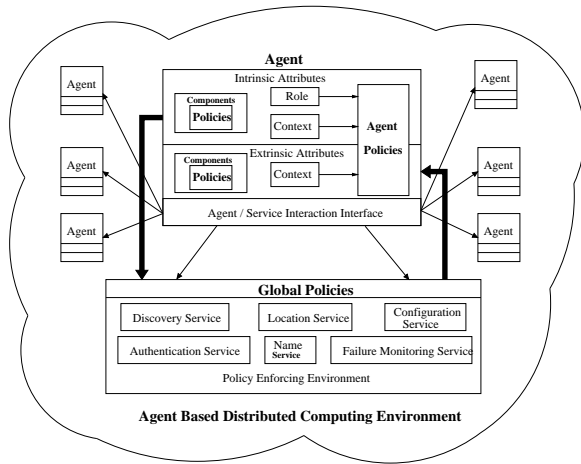


Figure 1: Agent immersed in the environment

at different locations in a network system. Events are also stored in databases for long-term correlations.

Secure distributed collaboration framework [11] supports construction of secure collaborative activities from their high level specifications. The specification of an activity defines a set of roles and a shared object space in which collaborating users perform their tasks. The specification model supports expression of the desired policies for coordination and dynamic security. Users participate in a collaboration through roles by executing operations (tasks) associated with the roles. A user's membership in a role represents a set of privileges, and these privileges are dynamically constrained by associating event-based preconditions with the role operations. Our middleware realizes a collaboration environment by creating for each role, object, and user an agent to enforce the desired policies. These agents communicate and interact with each other according to the policies derived from the activity's specification. The various different kinds of policies given to the agents in a collaboration environment pertain to role admission control, role operation preconditions, object access control, and event subscription-notification relationships among agents.

Section 2 discusses the central concepts in building of policy enabled agent-based distributed systems. In section 3 we discuss the core set of mechanisms required for policy-based architecture in agent based systems. In section 4 we use the two representative applications (Network monitoring, and Policy-based distributed collaboration) to elaborate on the policy framework and show its utility in developing real systems. In section 5 we compare the policies in network monitoring and distributed collaboration system and conclude in section 7.

2. POLICY FRAMEWORK FOR AGENTS IN DISTRIBUTED ENVIRONMENTS

Figure 1 shows an agent immersed in the environment and interacting with other agents and services in the environment. Associated with each agent are two kinds of attributes, intrinsic and extrinsic. Intrinsic attributes are agent attributes that are inherent to the agent. These include agent's role, its functionality and the associated components present in the agent, policies defining relationships

between different components and agent's behavior over time, and context information for the agent.

An agent acquires extrinsic attributes when it enters an environment. The extrinsic attributes of an agent include the external context information for the agent, information about other agents of specific attributes present in the environment, and policies regarding event subscription and notification from other agents and components. Whenever an agent enters the environment, it informs the configuration service of its presence along with its intrinsic attributes. Based on the intrinsic attributes, the configuration service configures the extrinsic attributes of the agent.

Each agent also consists of an interaction interface through which it can communicate with other agents and services present in the system. This interface is used for registration and delivery of events, registration of call-back handlers and remote deployment of new components within an agent.

Different services are available in the environment such as configuration service, discovery service, location service, failure monitoring service, name service and authentication service. The configuration service consists of a configuration database and the policy enforcement engine. This database is useful in finding out information about different agents and related policies in the system.

Policies are requirements and constraints over application level services, and relationships among distributed agents and their components. Policies are identified at various stages of service provisioning, from service design phase to service deployment phase. Enforcement of policies requires identification of rules outlining actions to be performed under different conditions. These conditions are characterized by events in our system.

In Table 1 we show different policy levels and the corresponding global and agent/component level requirements affected by these policies. Application/Service level policies include specifications of concerns such as which agents / components should be installed for the required services and functionalities, how many instances of various services are required and service-to-host mappings. They also include security policies based on agent's location, such as whether the agent is located behind a firewall or in a particular domain.

System's robustness policies include policies such as when to create/terminate/restart an agent/component, and the robustness requirements of various services. For example, one may specify when and where a service should be restarted if its host crashes. Policies may also specify creation of multiple service instances and their locations in the domain.

Corresponding to the application and system wide policies, agents are deployed in the domain. Such agents are driven by policies based on the agent's intrinsic and extrinsic attributes. Global application and system wide policies may require presence of certain component within each agent. Communication and coordination amongst distributed agents is required in order to realize application level policies, system policies, and other functionalities in the environment. Agent-interaction policies determine the set of agents with whom an agent with a given set of attributes should communicate and interact with. Policies may also enforce certain constraints on these interactions based on security requirements. For example, an agent may be restricted to communicate only with the agents residing in a specific domain or owned by some designated set of users. The interaction

Table 1: Policy Framework

Policy Level	Considerations at each level
Global Policies	Application/Service level requirements - which agents/components should be installed - service to host mapping System robustness requirements - when to create/terminate/restart an agent/component
Agent Interaction Policies	Agent's Intrinsic and Extrinsic attributes - role, context and components
Component Integration Policies	Component functionality - event relationship with other components - state restoration and recovery requirements - co-location of other components

relationships among agents, as determined by policies, are dynamic in nature as agent attributes and functionalities may change with time.

Agents contain application components and system components, deployed according to the application and service-level policies. These components generate events, and other local or remote components may be interested in such events. The robustness policies for the system may require that each agent contain a self-health monitoring component and report status events to a robustness monitoring service. All such application and system defined events will have to be sent by the encapsulating agent to remote agents and remote services. This requires inter-agent interaction. The agent's intrinsic attributes such as its location and security domain can also dictate its interactions with other agents. In all the above cases, agent interaction policies dictate the nature of communication relationship between agents.

Component policies describe the relationship between different components deployed within an agent. These policies include the list of local and remote events that will invoke the component, access to any system resources required by the component, and co-location of other components for proper functioning. These policies also relate to other component attributes such as what state of the component should be checkpointed, how often checkpoints should be taken, and if the component gets deleted then which other local and remote components should be informed about this. These policies can be defined at the component design time or can also be modified dynamically later.

3. SYSTEM LEVEL SERVICES AND MECHANISMS FOR POLICY BASED ARCHITECTURES

In this section we identify the functional requirements for a set of system services and mechanisms that are needed to support policy-based configuration and management of distributed agent system architectures. As new agents and components are deployed in such an environment, they need to self-configure according to the configuration policies. Similarly, when a component is removed from the system due to failures, administrative policies, or operational conditions (such as mobility of devices), appropriate reconfiguration actions are needed to be executed to preserve the invariants implied by the configuration policies.

The central concept in policy based configuration manage-

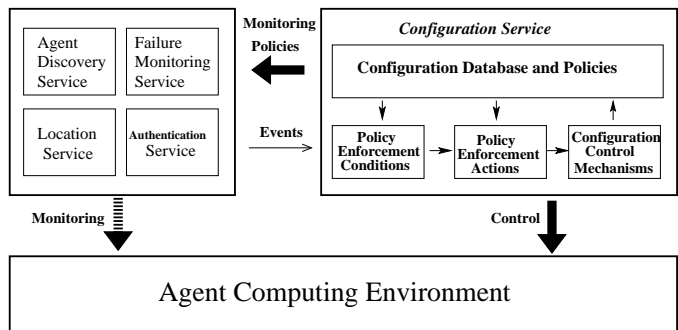


Figure 2: Policy Based Configuration Management Framework

ment is a reliable service that has knowledge of the global configuration policies and that is able to take suitable actions when configuration change events are detected. We refer to this as the *policy based configuration management framework*.

Figure 2 shows the different components of this framework. The framework consists of a set of *services* running in the environment and a *configuration service*. The configuration service knows the “configuration change events” that are considered critical for policy enforcement, and it instructs the services about its interest in the detection of such events. The function of different services is to detect these critical changes in the system environment and notify the configuration service of such changes. The configuration service is responsible for evaluating policies and enforcing them when critical changes occur.

One service is required for integrating new agents in the system. Its function is to detect the presence of new agents and notify the configuration service. This is called *agent discovery service*. In some architectures, the agents may first discover the configuration service and then inform it of their presence. Other services available in the environment are location service for detecting the locations of the users and an authentication service for providing tickets to the agents to enable secure resource access.

The *policies* for these two services are specified by the configuration service. For the *agent discovery service* there is a policy as follows: All the newly created agents must be known to the agent discovery service. This policy ensures

that when a device with its associated agents appears in the environment it is known to the other agents in the system.

The configuration service consists of a configuration database and the policy enforcement engine. This database is useful in finding out information about different agents in the system. It can be queried to find out all the agents having a particular attribute, a agent having certain set of attributes and other agent related queries. This database also stores the initial configuration information for different agents in the system. This can be used to restart an agent, if it is lost due to a failure. For example, it may list the set of components that must be installed on an agent at the time of restart and any specific initialization that needs to be performed for such components. This information is used in policy based failure recovery. An important design principle is to require each agent to perform its own recovery after it has been restarted, so the configuration service is rarely involved in maintaining any information about an agent's local state. Similarly, components within an agent should be designed to reconstruct their execution state, through checkpoints or other mechanisms such as "soft state".

The configuration service receives configuration change events when the *agent discovery service* informs it of the addition of new agents/components in the system, or the *failure monitoring service* informs it of any agent failures or departures. On receiving such events, the configuration service identifies the policies that depend on the change event and identifies the agents that are affected by those policies. It also executes the actions required for enforcing the policies. In some applications, the policy enforcement actions may be distributed among the the agents in the environment. In such cases, the configuration service notifies such agents. These actions ensure that the system state conforms to the requirements and constraints implied by the policies.

On detecting the failure of an agent, it may try to restart the failed agent on the same or a different host. In case of addition of a new or restarted agent, some of the other existing agents may need to be informed of its presence so they may establish appropriate interaction relationships with it. Similarly, the new agent is to be informed of the global policies that determine its functionalities and global interaction relationships with the existing agents. These policies may also require some new components to be integrated in an agent.

The above functions are performed by three different kinds of components that define policy enforcement conditions, policy enforcement actions and the configuration control mechanisms in the configuration service as shown in Figure 2. *Policy enforcement conditions* define the events that should trigger executions of the *policy enforcement actions*. A policy enforcement action first determines the set of agents and components that are affected by a given event according the policies. It also determines which new agents and components need to be created and installed in the system, and how the interaction relationships among the agents in the system needs to be altered. For this it may query the configuration database. Finally, it invokes the appropriate *configuration control mechanisms* to realize the necessary changes in the configuration.

Agents, and the components contained in them, may also explicitly register with the configuration service their interests in certain specific kinds of configuration change events based on their local policies. This requires the configuration

service to maintain such events of interest, called call-back information, for existing agents. This call-back information is also stored in the configuration database.

4. CASE STUDIES

In this section we provide two case studies demonstrating the utility of the policy identification framework for agent based distributed environments.

4.1 Network Monitoring System

We illustrate the usefulness of the policy framework of Section 2 by considering an application level requirement and a system level robustness requirement in our network monitoring system.

One of the application-level requirements in our system is to *detect abnormally high number of root login activities (successful/unsuccessful attempts) in the entire domain in a very short period of time*.

Using the policy framework of Section 2 we can identify the agents and components required to satisfy this requirement. An agent to monitor login activities on each host in the domain is required. This agent should contain a component to filter out root login attempts from all the other login attempts. There should be one agent present in the domain which will perform the root login correlation across the entire domain and flag a warning if the number of root login attempts goes beyond a threshold value.

The system level policy in Konark is that *all the agents in the system should be monitored for failure. Each agent should contain a heartbeat monitor component and should send AgentAlive events to the AgentFailure monitoring agent*.

Again using the policy framework we identify that each agent deployed in the environment should be provided with a heartbeat component. There should be at least one Agent-Failure monitoring agent always present in the system. Each agent should periodically send the heartbeat message to the AgentFailure monitoring agent.

Corresponding to the above application level policy and the system level policy, a newly deployed host is installed with a login monitoring agent with a login detection component, a root login filter component and a heartbeat component. The agent is also provided with the policies to notify the root login events to the root login correlator agent and the heartbeat events to the AgentFailure monitoring agent.

The extrinsic attributes of deployed components, policies about root login and heartbeat event notifications and the location contexts of the root login correlation and Agent-Failure monitor agents are driven by the global policies of the domain.

Arrival of a new agent in the environment also affects the policies of agents already present in the environment. The root login correlator agent and the AgentFailure monitoring agent have to subscribe to the respective events from the newly arrived agent.

Once an agent is deployed in the environment, the component integration policies are enforced. The component integration policies for the root login detection component at each host are shown in Figure 3. A periodic timer event triggers the execution of the *Syslog Event* detector. This detector generates events based on new log entries in the system log files. A *Syslog Event* triggers executions of the *Connection Event* detector, which filters and generates any login related events. This event triggers detectors for spe-

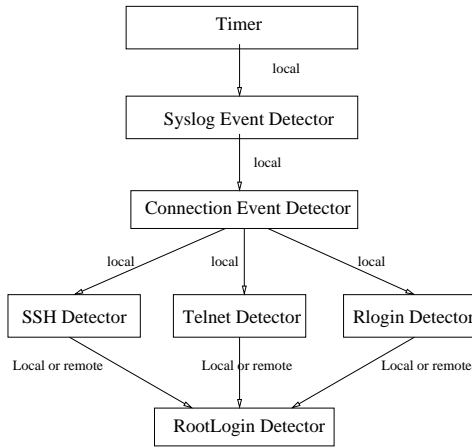


Figure 3: Component Trigger Dependency: Root Login Event Trigger Dependency

cific kinds of login, such as *SSH*, *Telnet*, and *Rlogin*. These detectors are filters which check if a given connection event belongs to a specific class. Any of these events, both local and remotely generated, trigger the *RootLogin* detector. 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. Similarly, a dependency marked only as *remote* implies that the triggering event detector must be in an agent different from the one containing the triggered detector.

For the AgentFailure monitoring agent, the component integration policy requires that the heartbeat component on the agent should only send the heartbeat message to an agent other than itself. Hence our system requires at least two AgentFailure monitoring agents to monitor **all** the agents in the system.

4.2 Secure Distributed Collaboration System

The secure distributed collaboration system defines a role based methodology for building collaborative applications from their high-level specifications. A collaborative application is built using the activity abstraction. An activity template consists of the template definitions of objects, services, roles, role-operations, operation preconditions, and role admission and activation constraints required for the collaborative application.

The configuration service, called **TrustedServer**, manages activity templates. Different activities can be instantiated from a given activity template. Each activity instance is managed by a separate *ActivityInstanceManager* agent. Each role within an instantiated activity is managed by a separate *RoleManager* agent. Users can participate in the collaborative activity through a user coordination interface (UCI) which is specific to each role in the activity. UCI is also modeled as a mobile agent thus making it device independent.

In a policy-driven collaborative application development there are policies for activity and role instantiation, role admission/activation and operation preconditions. Policy templates are specified at the time of activity specification. When an agent comes into the environment and informs the trusted server of its presence, the specific policy templates

based on the agent's intrinsic attributes are instantiated and given to the agent forming its extrinsic attributes. An activity agent is given the policy templates related to its child roles whereas a role agent is provided with the policy templates informing it of events that it should subscribe from other role agents.

Events are related to activities, roles, operations and services. Examples of events include instantiation of an activity, invocation of role operation, admission/removal of a user from a role, change of role member location etc.

```

Activity Meeting {
  ActivationConstraint
    currentTime > 8.00 am & currentTime < 6.00 pm
  Object room
  Object projector
  Object presentation
  Role Accountant {
    Operation DisplayFinancialData
    Precondition
      #(Chairperson.ApprovePresentation) = 1
      & room.isPresent(thisUser)
      & room.isPresent(member(Chairperson))
    Action projector.display(data)
  }
  Role Chairperson {
    AdmissionConstraint
      #(members(thisRole)) < 1
    Operation ApprovePresentation
    Action presentation.approve()
  }
}
  
```

Figure 4: Context aware collaboration activity

A meeting activity specification is shown Figure 4. The activity declares three objects (room, projector and presentation) and two roles (Accountant and Chairperson). All objects and roles are managed as agents. The activity is maintained in a template form at a trusted server and is instantiated when the meeting starts.

Based on the intrinsic attribute of its role, a role-agent is provided with extrinsic policy templates. These policies specify event subscription / notification from other roles and the service objects available in the activity.

The activation constraint policy for the activity specifies that the activity can be only instantiated between 8.00 am and 6.00 pm. The trusted server prevents activity agent creation if current time is not the correct time as specified in the *ActivationConstraint*.

The *DisplayFinancialData* in the Accountant role has a precondition requiring that the Chairperson role must have executed the *ApprovePresentation* operation and both the Accountant role member and the Chairperson role member must be present in the room. Hence when the Accountant role agent comes into the environment it subscribes to the events from Chairperson role agent if it is already present. If the Chairperson role agent is not present, it registers with the trusted server its dependence on the Chairperson role. It also subscribes to events from the room object to check the presence of users in the room.

The Chairperson role has an admission policy requiring that only one member can be present in the role. The Chairperson role agent performs this check and disallows presence of more than one member in this role. When the Chairper-

son role agent appears in the environment it is informed about the event subscription request from the Accountant role agent by the trusted server.

5. DISCUSSION

Both the network monitoring system and the distributed collaboration system have been implemented and are examples of closed world agent systems. Details of the design and implementation can be found in [15] and [11], respectively. Both these systems extend the Ajanta programming model with application specific policies. The policy based paradigm provides flexibility in building the inter-agent interactions and dynamically extending the functionality of the agents in both these systems. In fact, policies give the Ajanta agents the necessary autonomy in performing their tasks and managing their relationships with other agents and components in the system.

The nature of policies and the degree of agent autonomy differs between the two systems. In network monitoring system there are policies for configuration management of each detector within an agent, inter-agent communication and agent fail-over and restart. Agents have autonomy in deciding to whom they can report events, the rate at which events could be reported, restarting of a failed detector and maintaining event subscription and notification relationships.

On the other hand, the policies in the distributed collaboration system deal with event based security and coordination requirements between the participating role-agents and context based events [14] in collaborative applications. The enforcement of security and coordination constraints for a collaborative application requires that agents be supplied with strict guidelines about event generation and notification, limiting their dynamism as compared to the network monitoring system.

Policies help in specifying suitable tradeoffs for scalability of the network monitoring system at an expense of latency in detection of events of interest in the domain. For example, policies can be used to limit the reporting rate of successful root login events generated at a node to the correlator node with a possible delay in detecting whether the number of root login attempts in the domain is above a system defined threshold. The failure monitoring of the agents is also policy driven wherein an agent can designate its monitoring peers and keep on sending regular AgentAlive events to them.

6. RELATED WORK

The advantages of agent based architectures for building large scale software system are discussed in [1]. The application of agent based designs for building complex control systems are elaborated in [2] with specific concerns for diagnosis and repair functions needed in such systems to ensure robust operations. In [16] a three level model of organizational rules, organizational structures and organizational patterns is developed to address management aspects in open multi-agent environments. Our approach for policy based agent architectures reflects very similar kinds of notions with specific focus on inter-agent interactions, intra-agent component integration, and service level policies. Our approach provides a policy classification and autonomic mechanisms [4] targeted mostly towards closed domains.

The use of policies and their enforcement is the central

concept in our approach for building autonomic system architectures. Similar concepts have been proposed in the past by others in the context of *law governed systems* [6]. Policies can be viewed as a form of laws. Law based management of open multi-agent distributed systems has been studied in [7]. Our work embodies these concepts in a practical framework and illustrates their utility. Moreover, we identify the core set of services that are required for policy enforcement for autonomic configuration management. We also demonstrate here that agent architectures form an ideal basis for policy based systems.

The Konark system presented here provides mechanisms for distributed event monitoring and aggregation functions. It uses a subscription-notification model for communication among agents. This model is implemented using the underlying Ajanta agent communication model based on RMI. Other researchers have also developed similar event based systems for monitoring distributed systems [8] [5]. Our framework addresses more than just event notification and subscription. It presents an approach for autonomically establishing interactions among agents in response to configuration changes.

7. CONCLUSION

In this paper we have presented a framework for building large-scale distributed agent systems using a policy based approach. Policies represent requirements and constraints for integrating agent-based components in a distributed environment. This framework for policy based configuration management requires two essential services. One to enforce policies when certain critical changes in the system configuration occur, and the second is for monitoring the system to detect the critical changes. Autonomous agents provide an ideal foundation for a policy based approach.

Through two representative case-studies we have demonstrated the utility of this approach. Policy enforcement through a combination of mechanisms for event detection, handling, and subscription/notification provides a suitable abstraction for policy based component integration approach presented here.

8. REFERENCES

- [1] N. R. Jennings. An Agent-Based Approach for Building Complex Software Systems. *Communications of the ACM*, pages 35–41, April 2001.
- [2] N. R. Jennings and S. Bussman. Agent-based control systems: Why are they suited to engineering complex systems? *IEEE Control Systems Magazine*, 23(3):61–73, June 2003.
- [3] N. Karnik and A. Tripathi. Security in the Ajanta Mobile Agent System. *Software Practice and Experience*, 31(4):301–329, April 2001.
- [4] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, pages 41–50, January 2003.
- [5] M. Mansouri-Samani and M. Sloman. Monitoring Distributed Systems. *IEEE Network*, pages 20–30, November 1993.
- [6] N. Minsky and V. Ungureanu. Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM CM*

Transactions on Software Engineering and Methodology (TOSEM), 9(3):273 – 305, July 2000.

- [7] N. H. Minsky and T. Murata. On Manageability and Robustness of Open Multi-Agent Systems. In C. Lucena, A. Garcia, A. Romanovsky, J. Castro, and P. Alencar, editors, *Proceedings of Computer Security, Dependability and Assurance LNCS, No. 2940*, pages 189–206. Springer-Verlag, 2004.
- [8] J. C. Rowanhill, P. E. Varner, and J. C. Knight. Efficient hierarchic management for reconfiguration of networked information systems. In *International Conference on Dependable Systems and Networks (DSN'04)*, June 28 - July 01 2004.
- [9] C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley ACM Press, 1998.
- [10] A. Tripathi. Challenges Designing Next Generation Middleware Systems. *Communications of the ACM*, 45(6):39–42, June 2002.
- [11] A. Tripathi, T. Ahmed, and R. Kumar. Specification of Secure Distributed Collaboration Systems. In *IEEE International Symposium on Autonomous Distributed Systems (ISADS)*, pages 149–156, April 2003.
- [12] A. Tripathi, T. Ahmed, R. Kumar, and S. Jaman. Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 393 – 400, July 2002.
- [13] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh. Mobile Agent Programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 190–197, May 1999.
- [14] A. Tripathi, D. Kulkarni, and T. Ahmed. A Specification Model for Context-Based Collaborative Applications. *Elsevier Journal on Pervasive and Mobile Computing*, 1:21 – 42, May-June 2005.
- [15] A. R. Tripathi, M. Koka, S. Karanth, A. Pathak, and T. Ahmed. Secure Multi-Agent Coordination in a Network Monitoring System. In *Software Engineering for Large-Scale Multi-Agent Systems, 2002 (SELMAS 2002)*, Springer, LNCS 2603, pages 251– 266, 2003.
- [16] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational Abstractions for the Analysis and Design of Multi-Agent Systems. In *Agent Oriented Software Engineering*, January 2001.