

Design of a Policy-Driven Middleware for Secure Distributed Collaboration *

Anand R. Tripathi, Tanvir Ahmed, Richa Kumar, and Shremattie Jaman
Department of Computer Science
University of Minnesota, Minneapolis MN 55455

Abstract

We present here the execution model of a policy-driven middleware for building secure distributed collaboration systems from their high level specifications. Our specification model supports nested collaboration activities, and uses role-based security policies and event count based coordination specification. From the specifications of a collaboration environment, appropriate policy modules are derived for enforcing security and coordination requirements. A policy-driven distributed middleware provides services to the users to join roles in an activity, perform role specific operations, or create new activities. We describe here the design challenges for the middleware and present the runtime structures and protocols supported by it for creating activities, roles, and objects.

1 Introduction

In a Computer Supported Cooperative Work (CSCW) system, a group of users interact and collaborate using shared objects towards some common objectives [5]. Such systems range from real-time synchronous collaborations like online conferencing, interactive authoring of documents, to asynchronous workflow like collaboration in an office environment. Compared to traditional office automation workflow where activities are predefined, activities in real-time groupware systems are managed in an ad hoc manner [1]. The key issues in CSCW systems so far have been group awareness, multi-user interfaces, concurrency control, and coordination policies within the collaborating groups. However, security specification models for CSCW systems and the realization of the desired policies in an implemented system is a relatively new research area. Only few systems have addressed security policy specification models for collaboration systems; moreover, most of them have mainly concentrated on specific areas of CSCW like shared window based systems [4].

Our goal is to rapidly realize secure distributed collaboration systems from high level specifications comprising of security and coordination policies. A policy-driven system is able to specify the dynamic nature of CSCW activities and enforce coordination and security policies at runtime through a middleware. Existing policy driven CSCW systems, like COCA [8] and DCWPL [2], do not emphasize security issues in collaborative environments and are built mainly for shared interactive groupware applications. These systems are for smaller time-space limited activities, and therefore the models for describing the desired systems cannot implement large workflow like CSCW environments where multiple CSCW applications may come into existence within a single collaboration framework.

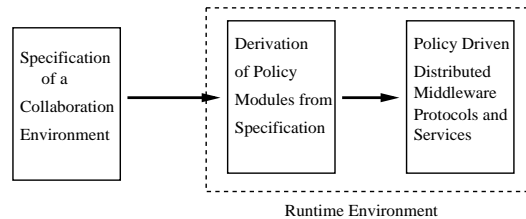


Figure 1. A framework for distributed collaboration systems

In our model, a policy-driven collaboration system is realized in three steps as shown in Figure 1. Initially, the coordination and security policy for a collaboration is specified based on a schema. From the specification, various policy modules are derived for different kinds of requirements, such as role based security, object level access control, and event notification for coordination. Finally, through these modules, the collaboration environment is realized by a generic middleware. As a first step towards our approach, we have developed a specification model [11] in which a collaborative system is defined in terms of activities, roles and objects. The model allows dynamic assignments of roles, “separation of duties” constraints, multiple user participation in a role, dynamic security policies, and hierarchical activity definitions.

* This work was supported by National Science Foundation grants ITR 0082215 and EIA 9818338.

This paper focuses on the design of a policy-driven middleware, which realizes the desired collaboration environment from its specification by generating policy modules and providing services and protocols for the runtime system. Our prototype implementation of this middleware is based on Java. The constraints and challenges in designing such a middleware are as follows:

1. All nodes within a distributed system cannot be trusted. Collaborating users work within their own local environments, which can be tampered with, and therefore policy enforcement and verification cannot be carried out at the user's end. However, within the system, certain nodes may be anointed as "trusted". Management activities and storage of shared data may take place at such nodes.

2. Each user in the system has a view of the policy, which is specific to his/her role. These views need to be securely distributed to the collaborating users. Due to the dynamic nature of the specification, these views may change at runtime and need to be appropriately updated.

3. Policies are required for secure storage of shared objects in a collaboration environment. Access control policies for an object can change based on context specific conditions for a collaboration. During the life cycle of an object, the trusted nodes for the object can change.

4. Roles within different activities represent different entities and need to be managed in their own domain of trust. Role management policies have to be established.

5. Operations performed by the users need to be securely communicated to other entities for coordination and security purposes. Enforcement of coordination conditions has to be delegated only to secure nodes.

A brief overview of our specification model is presented in Section 2 with a detailed example specification of a collaboration environment. Section 3 presents the challenges and constraints in designing a policy-driven middleware for secure distributed collaborations. In Section 4, we detail the distributed execution model of the system using the example from Section 2. Section 5 presents an overview of activity creation and view management protocols. Section 6 outlines how policy modules are derived. Section 7 and 8 discuss the related work and the conclusions.

2 Collaboration Specification Model

2.1 Overview of the Model

This section gives a summary of the specification model presented in [11]. In this model, an activity definition specifies a generic collaboration pattern among a set of roles using some shared objects. These entities are specified and named in nested scopes of activities as shown in Figure 2. An activity defines a protection domain and a scope for the roles, objects, and privileges in a collaboration. Importantly,

it works like a template and facilitates specification of dynamic or runtime behavior of a collaboration based on object types and roles. Any number of instances of it can be dynamically and concurrently instantiated. There are three major elements in specifying a collaboration activity: shared objects, roles, and operations. Shared objects

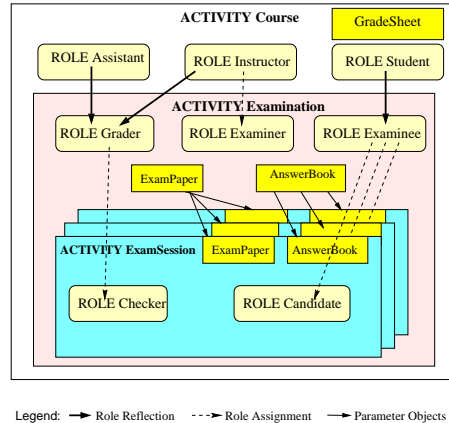


Figure 2. Hierarchical structuring of collaborative activities

are represented only in terms of their types and method signatures, keeping the semantics and implementation details transparent. We can specify access control at the granularity of the methods invoked on these objects. User level access control is specified in the scope of a role as part of its operation specifications. The object level access control specification is derived from various roles' privileges for an object.

A role operation may consist of a method invocation on a shared object, a synchronization action, or a role request. A role request is an operation to either *join* or *leave* a role. A synchronization action results in sending a coordination event to other entities.

In defining a role, we consider three types of constraints. The *role admission condition* puts constraints on admitting a participant to a role. The *role activation condition* must be satisfied when a role is activated. The *precondition* of an operation must be satisfied for a member in the role to execute the operation. In the specification model, a boolean function $member(role, user)$ checks if a participant is present in a role, $members(role)$ gives the list of participants in a role, and the number of participants admitted in a role is given by $\#(members(role))$.

Every activity instance has two meta roles: *Creator* and *Owner*. The user initiating the activity is admitted to the *Creator* role. The owner of an activity may be different from its creator and can be specified in the activity definition. If not specified, the owner of the parent activity

becomes the default owner. The owner of an activity also owns the roles encapsulated within its scope.

Coordination and dynamic security policies in our specification model are expressed using events and event counters based on the model presented in [9]. Related to each role operation are three types of events: *request*, *start*, and *finish*. For example, *role-name.op.start* or *role-name.op.finish*. These event types are also defined for each object method. The corresponding event counters are used for synchronization specifications. For a role, events are defined for *join*, *leave*, *admit*, and *remove* operations.

The multiple occurrences of a given event type are represented by a list. The expression *(eventName)* returns a list of all the instances of this type of event. Hence, *#(eventName)* returns the number of times the event has occurred. In our model, one can also specify a derived event type by filtering an event list based on its attributes. For example, for a role operation execution, we can define a filter based on invoker id, such as *opName.start(invoker=John)*.

2.2 An Example of Activity Specification

Using the *Course* example of Figure 2, we illustrate some of the key features of the specification model: hierarchical activities, creation of activity instances, and passing of objects and role participants from an activity to its child activity. In Figure 3, we describe the specification of the nested *Examination* activity shown in Figure 2. In our system, these specifications are expressed in XML.

An entity (such as an activity, object, object method, role, role operation or event) encapsulated in the scope of an activity can be referenced by a fully qualified name. Within an activity, one can refer to its current instance using the pseudo variable *thisActivity*, and its creator using *parentActivity*. The user executing an operation is identified by *thisUser*.

In Figure 2, a *Course* activity consists of an *Instructor* role, an *Assistant* role with possibly multiple teaching assistants as its members, and a *Student* role. In the *Course* activity, the *Examination* activity is defined as a nested activity. A participant in the *Examinee* role within this activity can create an *ExamSession* activity to take the examination. The participants in the *Grader* role are responsible for grading the students' answer-books.

A nested activity may need to have access to the objects in its parent activity, or the participants in a role in the parent activity may need to be admitted to a role in a nested activity. There are three ways in which users can be admitted to a role within an activity: *role reflection*, *role assignment*, or a member simply joins a role. Using role reflection, in an activity definition, a role in an activity can be statically assigned to a role in a nested activity. This means that all the members of the parent role become members of the role in

```

ACTIVITY Examination (OWNER Instructor,
                     ASSIGNED_ROLES Examiner) {
  OBJECT_TYPE ExamPaper (CODEBASE=http://codeserver) {
    METHOD setPaper {PARAM PaperType}
    METHOD readPaper {RETURN PaperType}
  }
  OBJECT_TYPE AnswerBook (CODEBASE=http://codeserver) {
    METHOD writeAnswer {PARAM AnswerType}
    METHOD setGrade {PARAM GradeType}
  }
  ROLE Examiner {
    OPERATION SetPaper {
      PRECONDITION #(SetPaper.start)=0
      ACTION { exam=new OBJECT(ExamPaper)
              exam.setPaper(data)}
    }
  }
  ROLE Examinee (REFLECT parentActivity.Student) {
    OPERATION StartExam {
      PRECONDITION #(Examiner.SetPaper.finish)=1
      & #(StartExam.start(invoker=thisUser))=0
      ACTION {obj=new OBJECT(AnswerBook)
              act=new ACTIVITY ExamSession(
                (exam,obj), Canadidate=thisUser)}}
  }
  ROLE Grader (REFLECT parentActivity.Assistant,
              parentActivity.Instructor) {
  }
  ACTIVITY ExamSession( OWNER Grader,
                       OBJECTS (Course.ExamPaper exam,
                                Course.Examination.AnswerBook ans),
                       ASSIGNED_ROLES Candidate) {
    TERMINATION_CONDITION: #(Checker.Grade.finish)>0
    ROLE Candidate {
      ADMISSION_CONSTRAINTS
        member(thisUser, parentActivity.Examinee)
        & thisActivity.creator=thisUser
        & #members(thisRole)<1
      ACTIVATION_CONSTRAINTS
        date > DATE(Jan, 12, 2001, 12:00)
        & date < DATE(Jan, 12, 2001, 15:00)
      OPERATION Read {
        ACTION exam.readPaper()
      }
      OPERATION Write {
        ACTION ans.writeAnswer(data)
      }
      OPERATION Submit {
        PRECONDITION #(Write.finish)>0
      }
    }
    ROLE Checker {
      ADMISSION_CONSTRAINTS
        #members(thisRole)<1
        & member(thisUser, parentActivity.Grader)
      OPERATION Grade {
        PRECONDITION #(Candidate.Submit.finish)=1
        ACTION ans.setGrade(data)
      }
    }
  }
}
}
}

```

Figure 3. Specification of an Examination

the child activity, at the time of activity creation, subject to its admission constraints. Removal of a participant from the reflected role (i.e. a role in the parent activity), also implies removal from the role in the child activity. A participant in the reflected role gains expanded privileges comprising of the operations of the child activity role. In our example, both the *Instructor* and the *Assistant* roles in the *Course* activity are reflected in the *Grader* role of the *Examination* activity. When an instance of this activity is created, the members of these two roles are admitted to the *Grader* role. Therefore, an admitted participant in the *Grader* role can perform operations on the *GradeSheet* object in the *Course*

activity as well as on the objects in the *Examination* activity. Similarly, the participants in the *Student* role are reflected in the *Examinee* role.

Participants can also be dynamically assigned to a role as part of activity creation. The `ASSIGNED_ROLES` tag in an activity definition specifies the roles that need to be assigned during activity creation. For example, as specified in Figure 3, participants for the *Examiner* role have to be assigned at the time of an *Examination* activity instantiation. Also, the owner of a role can assign participants to that role. For example, when the *ExamSession* activity is created by an examinee, the *Grader* role becomes the owner of that activity and the roles within its scope. The *Grader*, being the owner of the *Checker* role, may assign a participant to it.

In Figure 3, a student in the *Examinee* role can initiate an examination session by invoking the *StartExam* operation. However, he/she cannot initiate two sessions of this examination as enforced by the precondition. The *StartExam* operation starts an *ExamSession* activity. The *ExamSession* activity contains the roles *Candidate* and *Checker*. Only the student creating this activity is assigned to the *Candidate* role. References to two objects, an *ExamPaper* and an *AnswerBook*, are passed as parameters to an exam session activity. A single *ExamPaper* object is shared by all the sessions. On the other hand, a new *AnswerBook* object is created for each student's exam session. This *ExamSession* activity terminates when the checker finishes grading the answer book. The activation constraints ensure that the examination must be taken during the specified three hour period.

A candidate can perform operations to read the exam paper and write the answer book. The *Submit* operation can only be performed when something is written. However, *Submit* does not invoke any method, rather it creates an event subscribed by the *Checker* role for coordination. The checker can only grade after the answer book is submitted. The cardinality of the *Checker* role is one, i.e., only one checker can be assigned to an answer book.

3 Middleware Design Challenges

The distributed architecture and security considerations of the policy-driven middleware impose several design challenges, as discussed below.

Policy module creation and distribution: Several different kinds of policy modules are needed to manage an activity. For a role in an activity, we need a policy module containing admission and activation constraints. It should also contain preconditions for role operations together with the directives for subscribing events from other entities in the system for evaluating these preconditions. For an object, a role-based access control list is required by its object server. This access control list should identify the method invoca-

tion privileges that should be given to a role in a specific activity instance, together with the role operation context in which a method is invoked. Because an invoker may not be trusted, the object server should also verify the operation's precondition. For an activity template, we need a policy module defining the activity creation privileges to be given to certain roles. An activity instance can have templates for nested activities, which need to be made visible to the participants in various roles within that instance. For example, the *Examination* activity has the *ExamSession* template.

From an activity definition, we need to build templates for policy modules. When an activity instance is created, the corresponding policy templates also need to be instantiated using context specific information pertaining to the new instance. These modules then need to be communicated to the appropriate roles and object servers.

Secure sites for policy enforcement: The creator of an activity may not always be trusted with the responsibility for that activity's management. In the *ExamSession* activity example, a student in the *Examinee* role creates an *ExamSession* activity. However, the student's node cannot be trusted to enforce security policies for the *ExamSession* activity, as a student can change and manipulate the activity specific policies. Only trusted secure sites for a role should be able to maintain its membership information and enforce role membership operations. Similarly, only trusted sites should control who can create an activity instance from a template.

When there are multiple participants in a role, we need mechanisms to ensure that preconditions for operations are evaluated and enforced consistently and securely. Consider the precondition for the *Examiner* role's *SetPaper* operation, which requires that the *SetPaper* operation must not have been started by anyone else. Initially this precondition is true for all participants; therefore, we need a mechanism so that only one of them starts this operation.

Distributed trust relationship among roles: Users participate in a collaboration through roles in the context of activities. The participants may need to be uniquely identified for inter-role coordination, i.e., the participant may need to be uniquely identified in the context of a role. In some cases participants are admitted to a role based on their role in a parent activity, like the *Grader* role in our example. If membership certificates are used for enforcing such admission conditions, a trust relationship needs to be maintained among participating sites to enforce the authenticity of such certificates. Different situations may warrant different revocation policies, which could be *immediate* or *delayed*. In the *ExamSession* activity, a user can be given a *Candidate* role certificate, which can be used once per session. Though the certificate is based on the user's previous *Student* role certificate, the *Candidate* role certificate need not expire even if the *Student* role certificate expires. On the other hand, the

invalidation of the reflected *Assistant* role certificate has to be propagated immediately to the *Grader* role.

Object storage and protection: The stages of a workflow can change ownership and other access control policies of an object. Arbitrary storage location of an object cannot be trusted. If a trusted site is specified during object creation, this site may not be trusted at a later stage of the workflow. In our example, the student in an *ExamSession* activity creates an *AnswerBook* object and should be the owner of this object. However, after submitting the answer book, the student should not be able to make modifications, and the ownership has to be transferred to the *Grader* role. The owner of an object can impose additional discretionary access control on the object.

Various versions of an object may reside at different locations. Each object needs to maintain and enforce its access control policy. A subject needs to be uniquely identified by its activity, role, and often by user identity certificates. Similar authentication is required for modifying the policy object itself.

Enforcing preconditions of an operation only at the users' nodes is not secure and must be enforced at a secure site where the object is stored. For example, enforcing the precondition `$(Candidate.Submit.finish)=1` of the *Grade* operation only at the *Checker* role's node does not guarantee consistency. It is possible for a malicious user to tamper with execution environment at his/her node and invoke an object method even when the precondition for the corresponding operation is not true. Therefore, for each method invocation we require the object server to verify the invoker's id, role, and the name of the role operation in which this method is being invoked. It should then verify the operation precondition and access rights.

Secure distributed event service: The coordination and the dynamic security policies are specified as preconditions or constraints based on event counts. Our approach requires event subscription-notification policy modules to be derived from the specification. Additionally, event integrity and secrecy need to be guaranteed. Only authorized entities should be able to generate or receive an event.

4 Distributed Execution Model

Every activity in our model is represented as an object tree structure. This tree structure representation is used both for system-level management of an activity as well as for building user-level interfaces for the participants. A node in the object tree of an activity represents a nested entity such as an activity template, an activity instance, an object type, an object instance, or a role. A node representing a nested activity forms a subtree within its parent activity's object tree. The primary function of a node is to provide access to management functions for the entity that

it represents. For example, the node for an activity template supports functions for creating new instances in compliance with the security policies, and keeping track of all instances of the activity. For each instance, it maintains child node. A node corresponding to a role is responsible for admitting/removing users from the role and enforcing admission constraints when a user joins a role. It also provides to the participants interfaces for invoking role-specific operations subject to coordination conditions.

Object type nodes have functions similar to the activity template managers, they decide which roles can create an instance of that type, and maintain references to all the object instances that are created. Nodes representing object instances are responsible for object storage, caching/replication management, and authenticated invocation of its methods. The object tree graph of a collaboration may change at runtime as new activities or objects are created.

In a distributed environment this tree structure is partially replicated to provide each participant a view of the collaborative environment based on his/her role-based privileges. We refer to this as a *role-specific view*. A node representing an entity in the object tree can be visible to multiple participants. One node among all these nodes, which is maintained at a trusted site is designated as the *authoritative* or *primary node* for that entity. A primary node is responsible for managing all security sensitive functions and event subscriptions for the entity it represents. For example, an object instance's persistent storage is maintained at an object server, which is trusted by the user having the primary node for it. All other users which see that entity have a *stub node* to communicate with the primary node. The stub node contains the protocols for authenticated invocation of functions at the primary node. In some cases, the primary node of an object may support caching of the object by its stub nodes. Also, each primary node maintains references to the other copies (which are stub nodes) for performing any callback functions, for example, role revocation and event communication.

The system provides each user with a coordinator on his/her system, the *User Coordination Interface (UCI)*, which maintains the user's view of the object tree and provides suitable interfaces to the user. Whenever a user joins the environment, he/she first sees only a default root-level activity, termed *System*. To define a new collaboration system, a user in the *Convener* role in the *System* activity develops an activity template for it and installs it under the *System* node. A user can list all the activities that have been defined under *System* and that he/she can join. The primary node for the activity is located at the owner's UCI. It is essential that the owner of an activity be trusted. The default owner of any activity is the owner of its parent activity, unless otherwise specified.

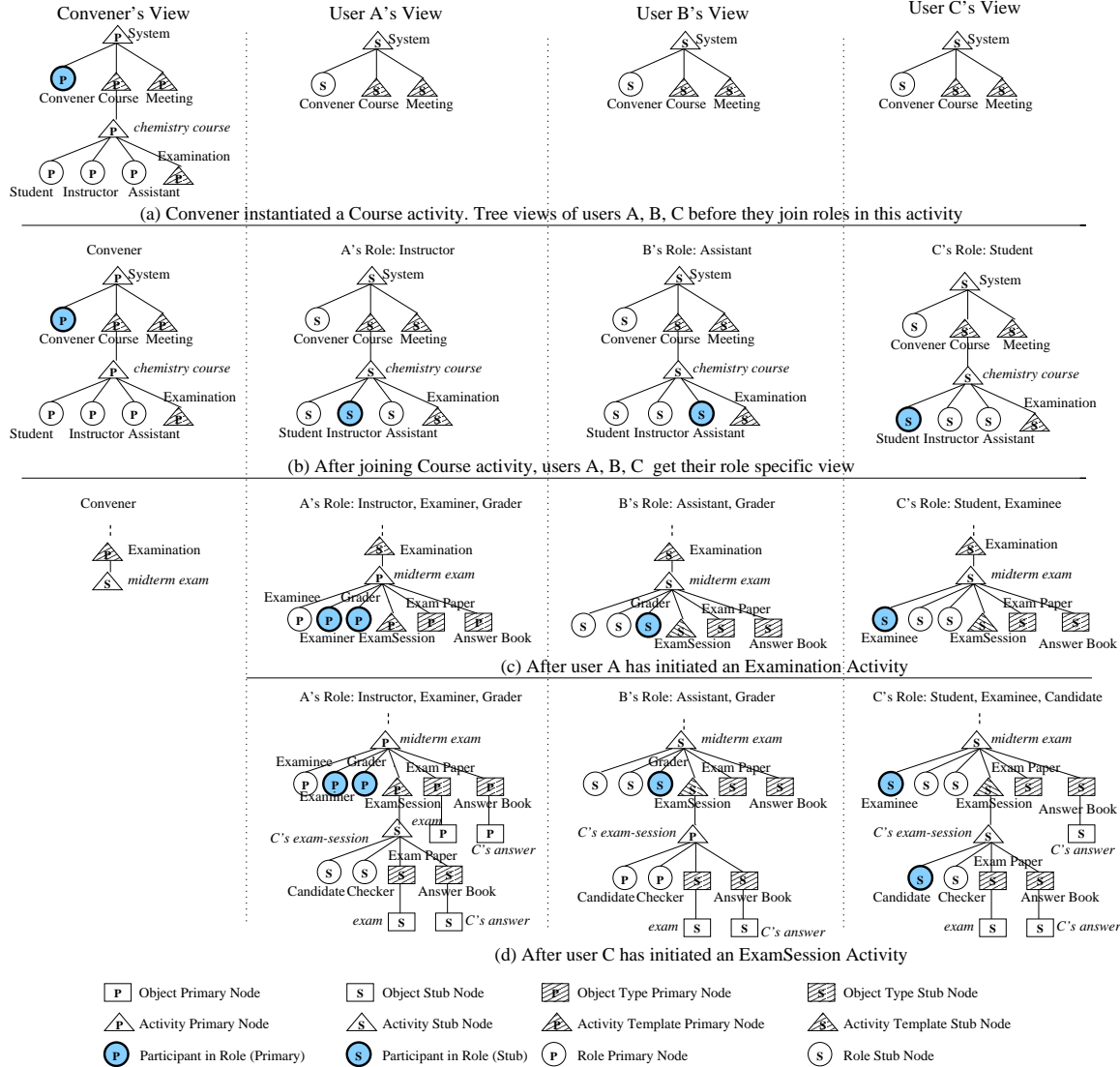


Figure 4. System level view of a distributed collaboration environment

Figure 4 illustrates the execution model for our example, which shows the role-specific views for a convener and users A, B, and C. This illustration uses twelve kinds of nodes in the object graphs at different users' sites. There are two kinds of nodes — primary and stub – for each of the four objects: activity template, activity instance, object type, and object instance. For a role, we have primary and stub nodes, and we use shading to indicate if a participant is a member in that role.

In Figure 4(a), the convener has created an instance of the *Course* activity, *chemistry*, the other users are not aware of this activity instance yet. By default, the *Convener* is the owner of the *chemistry course* activity, and the primary nodes of this activity reside at the *Convener's* site. Assuming that user A is assigned to the *Instructor* role, user B

to the *Assistant* role, and user C to the *Student* role, the role specific views of the *chemistry course* activity are distributed to the corresponding users' UCIs as shown in Figure 4(b).

According to the specification, the *Instructor* can instantiate an *Examination* activity through its stub node and is the owner of that instance. In Figure 4(c), user A in the *Instructor* role has created an instance of the *Examination* activity *midterm exam*. Users A, B, and C are assigned to roles in this activity by role reflection. Their views get updated by the owner of the activity. User A is now a participant of the *Instructor*, the *Examiner*, and the *Grader* roles. Being the owner of the *midterm exam* activity, user A's site maintains the primary nodes of this activity's roles, *ExamSession* template, and object types: *ExamPaper*, *AnswerBook*.

In Figure 4(d), the *Examiner* creates an *ExamPaper* object *exam*, and user C instantiates an *ExamSession* activity with an *AnswerBook* object, *C's answer*. *C's answer* is created at the *Instructor's* site, as this role is the owner of the activity *midterm exam*.

5 Activity Creation and View Management

The middleware provides protocols for object method invocation, activity instantiation, role specific view distribution, and role management. In this system, several existing middleware services of Ajanta [10] for naming, public key maintenance, and secure method invocation are utilized. Events are securely communicated through authenticated RMI between publishers and subscribers.

Object methods are invoked by participants as part of role operations. The primary node managing an object authenticates the invoker based on role *membership* and *ownership* certificates. In our distributed trust model, a certificate is signed by the owner of the certified entity's parent activity. In the example in Figure 4(d), the user C in the *Candidate* role performs the *Write* operation, which invokes the *writeAnswer* method on the *C's answer* object through the stub node. The primary node for *C's answer* authenticates the participant using his/her cascaded certificates. User C presents the *Candidate* role member certificate signed by the *Grader* role of *midterm exam*. Attached with this is the *Grader's* ownership certificate for the *Candidate* role certified by the *Instructor* role in *chemistry course* activity. This proves that the *Grader* is authorized to certify the *Candidate* role membership. Considering that the *Instructor* owns the *C's answer* object, a certificate by the *Instructor* can be trusted for this method invocation.

To create an activity a request is sent to the primary node of the activity's template. If the request is allowed by the access policy, it is passed to the owner of the activity. The owner of the activity creates the object tree for this new activity and attaches it in its view just below the stub node of the activity's template. This subtree contains the primary nodes for all roles, objects, and activity templates in the new activity. The primary node of this new activity's template is notified and a stub node for this activity is attached to the template's primary node. Moreover, the role specific view of this activity is now sent to the UCI of each participant that is present in any role within this new activity. Each recipient attaches in its view this tree for the new activity below its template's node. In our example in Figure 4(d), user C in *Examinee* role requests the creation of an *ExamSession* activity through the stub *ExamSession* template. This request is sent to user A's site, which maintains the primary node of the *ExamSession* activity. In the *ExamSession* specification, the *Grader* role is designated as its owner. Therefore, one of the participants of the *Grader* role, user B in Figure 4(d),

is chosen to be the owner and to maintain the primary nodes for this activity *C's exam-session*.

When a new activity is created, the primary node for each role within this activity obtains from the reflected roles the list of all participants. It also sets up callback at the reflected role's nodes to get revocation notifications for any of the reflected members. The owner now sends to each of the participant's UCI a request to update its view. The request contains the owner's ownership certificate (for authentication), the recipient's role certificate, the subtree representing the new activity, and the exact location within the participant's view where this subtree needs to be attached.

6 Policy Module Derivation and Distribution

Every primary node maintains various policy modules for role management, access control, and event subscription and notification. These policy modules are initially derived from an activity template and are modified when instances of the activity are created. Moreover, policy modules are specified as *policy templates* in the context of activity instances, i.e, a policy is specified based on activity types, roles, and object types. The access control policy template for an *ExamPaper* object instance is shown below:

```
OBJECT=ACTIVITY(Course, x).ACTIVITY(Examination, y)
                                     .OBJECT(ExamPaper, z)
OWNER=x.Instructor
ENTRY{
  SUBJECT=y.Examiner
  PERMISSION=setPaper
  CONDITION=(#(y.Examiner.SetPaper.start)=0)}
ENTRY{
  SUBJECT {ACTIVITY_TEMPLATE=y.ExamSession, ROLE=Candidate}
  PERMISSION=readPaper
  CONDITION=null }
```

The *ExamPaper* object instance *z* is specified in a nested context of a *Course x* and an *Examination y* activity. Object *z* is owned by the *Instructor* role in *Course x* activity. The first entry specifies that the *Examiner* in the *Examination y* instance can invoke the *setPaper* method on object *z* when the specified conditions are satisfied. This event condition is subscribed from the primary node of the *Examiner* role of the *Examination y* instance. In the second access control entry, the policy for invoking *readPaper* method on object *z* can only be realized when a *Candidate* role is created in an instance of the *ExamSession* activity template. This privilege is specified using the context of an activity template and a role. As new instances of activities, roles, or objects are created, binding takes place in the above policy template. The resultant policy templates are distributed to the primary nodes of the newly created entities. As activities and objects get instantiated at every stages in Figure 4, the above template becomes more specific to the object, and finally fully instantiated when the *exam* object is created. The final access control policy module for the *exam* object is shown below:

```

OBJECT=Course.chemistry.Examination.midterm.ExamPaper.exam
OWNER=Course.chemistry.Instructor
ENTRY {
  SUBJECT=Course.chemistry.Examination.midterm.Examiner
  PERMISSION=setPaper
  CONDITION=(#(Course.chemistry.Examination.midterm
               .Examiner.SetPaper.start)=0)}
ENTRY{
  SUBJECT {ACTIVITY_TEMPLATE=Course.chemistry.Examination
           .midterm.ExamSession,
           ROLE: Candidate}
  PERMISSION: readPaper
  CONDITION: null}

```

Similarly, other policy template modules are installed with the primary nodes managing the corresponding entities.

7 Related Work

Our work is similar to COCA [8] and DCWPL [2] in their approach of constructing a distributed collaboration environment from a high level specification. COCA [8] is a logic-based coordination policy specification language for interactive CSCW applications. At the implementation level, COCA is strongly tied with IP multicast models and Prolog, which may not scale for a large collaboration. DCWPL [2] addresses user level mechanisms to deal with group interaction issues and is limited to its predefined policies and functions. Neither of these approaches support dynamic security policies, like policies for dynamic “separation of duty” constraints. These approaches can only specify coordination policies based on previously executed operations and cannot independently specify discretionary access control for an object. Several other systems, such as CSDL [3], mainly concentrate on supporting various floor control policies in shared workspace environments. Compared to these, our specification model supports a wide range of CSCW systems. A distributed and decentralized management of roles in a distributed service model is presented in [7]. In our model, we are concerned with an integrated specification of collaboration environments using activity templates and distributed realization of such activities. Our goal is to derive an implementation from a specification.

Our approach is based on a composition schema to integrate a collection of objects to build a collaboration environment. This approach is akin to aspect-oriented programming [6] in terms of separating security and coordination concerns from the design of collaboration objects. However, our work also demonstrates that some aspects of security concerns are tightly coupled with coordination requirements. Our middleware supports aspect management by integrating policy-driven wrappers with collaboration objects.

In [10], we investigated a mobile agent-based collaboration system. The primary objective of that work was to investigate the use of mobile agents for object mobility in a middleware for supporting distributed collaborations.

8 Conclusion

We have presented in this paper a distributed execution model for secure collaboration systems. We use this model in a policy-driven middleware for supporting rapid construction of a collaboration environment from its specifications. The specification model presented here supports role-based security policies and nested activities. The activities in a collaboration are managed in a decentralized manner based on a distributed trust model to enforce the required security policies and coordination constraints. In our approach, policy modules are derived from the specifications of a collaboration activity. The policy modules are related to the management of activities, roles, objects, and event communications in the collaboration. These policy modules are distributed to different users’ computing sites based on the security requirements and the user’s role. We have presented our design using a detailed example of a course examination activity.

References

- [1] J. Bardram. Designing for the Dynamics of Cooperative Work Activities. In *Proc. of CSCW’98*, pages 89–98, 1998.
- [2] M. Corts and P. Mishra. DCWPL: A Programming Language for Describing Collaborative Work. In *Proc. of ACM CSCW’96*, pages 21 – 29, November 1996.
- [3] F. DePaoli and F. Tisato. Cooperative Systems Configuration in CSDL. In *Proc. of the 14th IEEE Intl. Conference on Distributed Computing Systems*, pages 304 –311, 1994.
- [4] P. Dewan and H. Shen. Access Control for Collaborative Environments. In *Proc. of ACM CSCW’92*, pages 51–58, 1992.
- [5] C. A. Ellis, S. J. Gibbs, and G. Rein. Groupware: Some Issues and Experiences. *Communications of ACM*, 34(1):39 – 58, January 1991.
- [6] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming: Introduction. *Communications of the ACM*, 44(10):29 – 32, Oct. 2001.
- [7] R. Hayton, J. Bacon, and K. Moody. Access Control in an Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3 –14, 1998.
- [8] D. Li and R. Muntz. COCA: Collaborative Objects Coordination Architecture. In *Proc. of CSCW’98*, pages 179–188, 1998.
- [9] P. Roberts and J.-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proc. of IFIP Congress*, pages 981–986, 1977.
- [10] A. Tripathi, T. Ahmed, V. Kakani, and S. Jaman. Distributed Collaborations using Network Mobile Agents. In *Proc. of ASA/MA, Springer-Verlag LNCS 1882*, pages 126–137, Sept. 2000.
- [11] A. Tripathi, T. Ahmed, and R. Kumar. Specification and Implementation of Secure Distributed Collaboration Systems. Technical report, Dept. of Computer Science, Univ. of Minnesota, Sept. 2001. TR 01-039.