# Secure Management of Role Based Distributed Collaboration Systems

*Author Name*

Author Affiliation
Street Name
City

**ABSTRACT**

This paper presents secure protocols for runtime management of activities in role-based distributed collaboration systems constructed from their high level specifications. The system architecture and the protocols presented here take into account two different aspects of the security requirements: collaboration specific requirements, and system level requirements to manage the distributed runtime environment. We identify different modes of interactions among the various entities in a collaboration system – users, roles and objects – and develop protocols for their secure execution under distributed trust relationships.

## 1 INTRODUCTION

In computer supported cooperative work (CSCW) systems, multiple users cooperate and collaborate using shared objects to achieve some common objectives [8]. Examples of CSCW systems include synchronous online conferencing, product design and development, authoring of documents, health care activities, asynchronous workflow in an office environment, and collaboration among different organizations. Due to the dynamic nature of collaboration requirements, policy based approaches have been proposed for CSCW systems [6, 15, 3]. Most of these systems mainly support coordination policies to enforce synchronization and precedence constraints on user activities. However, CSCW systems need to support security policies related to secrecy and integrity of the shared data, confidentiality, auditability and non-repudiation of users' actions, and integrity of operations to ensure that only authorized participants perform certain sensitive tasks.

With the advent of communication technologies, current collaboration systems are inherently distributed as users from different administrative domains may need to cooperate. Even within a single domain, participating users can be competing. It may not be possible to trust a single domain or site with the management of all aspects of a distributed collaboration. Moreover, from the security point of view, centralized solutions introduce a single point of attack. Managing a policy based distributed collaboration introduces challenging runtime security requirements for interactions among the collaborating users and the system.

The primary objective of our research is on developing techniques for constructing secure distributed collaboration environments from their high level specifications. We have developed a role based specification model capturing coordination and security requirements [Reference removed for anonymity]. Based on the specification, we derive the runtime middleware to enforce the security and coordination policies by the distributed collaboration entities. We presented the distributed runtime execution model in [Reference removed for anonymity, to appear in an IEEE conference]. Only few systems, such as COCA [15] and DCWPL [3], have taken this kind of approach of building a collaboration environment by coupling its high level specifications with a middleware. In contrast to our work, these systems do not have a security model for supporting meta-policies for role management, discretionary access control for an object, or active security policies like dynamic "separation of duty" constraints. Another limitation of these systems is that all user nodes are equally trusted.

The focus of this paper is on the design of middleware protocols for securely managing activities, roles, shared objects, and user interactions in a policy based distributed collaboration system. The objective of this work is to support construction of such secure collaboration environments. In this paper, we discuss the important security requirements for securely managing the runtime system and facilitating secure user interactions. In the context of the policy driven middleware, following are the main contributions:

- An ownership delegation model based on a trust relation which allows the runtime system to determine which sites can be trusted to manage specific tasks.
- The protocols for secure interactions among users, role managers, and object managers based on different paradigms of interactions between a user and shared objects.

The following section presents our role-based collaboration model, identifying different interaction models in a collaboration. Section 3 presents the security requirements and challenges in managing the runtime functions of a collaboration system. Section 4 discusses owner assignments for managing distributed collaboration entities. Section 5 gives an overview of the runtime execution environment and describes the basic middleware services. Section 6 gives the protocols for secure interactions within the system. Finally, we present the related work and conclusions.

## 2 A ROLE BASED COLLABORATION MODEL

In a collaboration, a group of users cooperate and coordinate using shared objects towards some common goal. Based on the responsibilities and privileges within a collaboration, many collaboration systems [23, 4, 8, 11] identify users' roles to specify coordination and security policies. Roles facilitate the specification of these policies for collaboration systems without knowing the users who will participate in the collaboration.

In a collaboration environment, users may be distributed over a network, and they may be associated with different administrative domains. In our model, a user participates in a collaboration by joining a role. An action performed by a user through a role can result in a sequence of interactions between the invoker and a set of shared objects. Each user's execution environment consists of a generic facility called the *User Coordination Interface (UCI)*, which provides to the user mechanisms for participating in a collaboration. Basic entities of a role based collaboration model are shown in Figure 1.

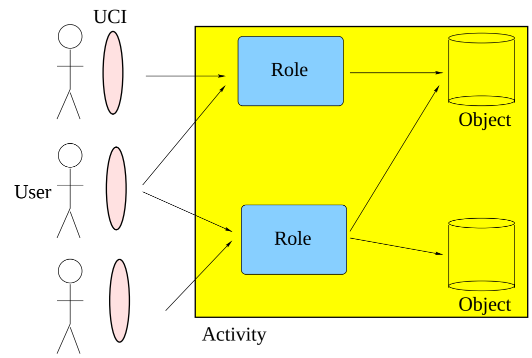


Figure 1: Entities in a role based collaboration model

In our collaboration model, an activity defines how a group of users cooperate towards some common objectives by performing their individual tasks on a set of shared objects. It represents a protection domain and a scope for the roles, objects, and privileges in a collaboration. In an activity, users are represented by their roles, and roles are assigned privileges to perform certain tasks. We term these role specific tasks as *operations*. Operations can be method invocations on shared objects, synchronization actions, or activity management actions. Operations may have preconditions to coordinate users' actions. An activity can be structured hierarchically, consisting of multiple nested concurrent activities.

The collaboration model that we have developed and adopted in our work has been motivated to support different forms of interactions between users, roles, and objects. The modes of interactions are outlined below. Moreover, this model supports specification of coordination requirements and security policies in regard to management of roles, objects, and hierarchically structured activities. Following is a brief description of the design requirements for this model. Further details of this model and its specification language can be found in [Reference removed for anonymity].

### 2.1 Interaction Models

In our collaboration model, coordination and security policies are specified using roles. As all tasks within our collaboration system are modeled as role operations, a user may not need to know which methods are invoked as part of a role operation. Consistency of the collaboration system depends on the coordination states of the roles. To maintain the consistency of roles' internal states in the execution environment, we impose the restriction that all the interactions among users and objects must be initiated through a role. A role operation can result in multiple method invocations. These method invocations can be on a single or multiple objects. In such cases, an adapter object is constructed to handle issues related to multiple method invocations.

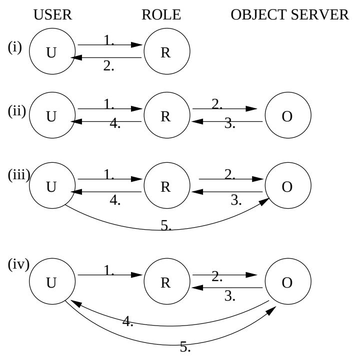

Figure 2: Interaction models

The users participate in a collaboration through UCIs, and roles and objects are managed on trusted servers. Different interaction models that emerge among these entities within our collaboration model are shown in Figure 2. In (i), a role operation is invoked by a user only for coordination reasons, and no object method needs to be invoked. Completion or failure of this coordinating operation is notified to the user. In (ii), the role invokes object methods on behalf of the user. In many cases, a user may require multiple interactions with a single or multiple objects, which is initiated through a role operation. Such cases are illustrated in (iii) and (iv), when a role operation may result in establishing a session for direct interactions between the user and the object. The object server sets up a session, and the objects' references and session information are passed to the user either through the role, as in case (iii), or directly using mechanisms like transportable mobile objects/agents, as in case (iv).

### 2.2 Role Management

The security policies need to determine who can be admitted in a role, who can admit a user to a role, when a role specific access right can be exercised, how many members should be present in a role, or which roles a user is allowed to join or activate concurrently. Role admission constraints are required to specify the conditions that need to be satisfied before a user can join a role. These constraints can be based on: a list of users allowed to join a role, a list of those who should never be admitted, previous qualifications, role membership cardinality, or occurrence of certain events. Many of these role management requirements have been addressed by research in role based access control (RBAC) [20] and policy based system management [17].

In this model, preconditions are associated with role admission constraints, role operations, and activity termination. An operation execution request is allowed to proceed only if its precondition is true. The preconditions are used for enforcing coordination constraints and dynamic security policies. The preconditions are expressed as event counts using the model presented in [19]. Events correspond to instantiation of activities, execution of role operations, admission of users in roles etc. In our model, related to each role operation are two types of events: *rolename.opname.start* and *rolename.opname.finish*. The *start* event for an operation indicates beginning of its execution, and the *finish* event signifies its completion.

Meta roles, such as *Owner*, *Creator*, need to be defined, which can admit users in a role, deactivate a role, or perform other administrative tasks on the role.

Privacy can also become an issue when one may need to hide the identity of one participant from another. In such cases, the presence of a participant may only be visible through his/her role or a pseudonym in a role but not by name. Privacy requirements of group members and group protocols are addressed in [5].

### 2.3 Object Management

In a collaboration environment, shared objects must be stored on trusted and reliable object servers. The traditional concerns of confidentiality and integrity of shared data are naturally present in collaboration systems. However, access rights, privileges, and ownership of objects may change in collaborative environments as activities progress.

Objects may move from one object server to another or from one collaboration to another, possibly across administrative domains. In such cases, security requirements may change based on the object's location, and access control policies may need to be modified at runtime. Moreover, when an object needs to be shared in a newly instantiated collaboration, its access control policies may need to be updated by authorized entities. If access control depends on coordination constraints, the object manager may need to verify operation preconditions.

### 2.4 Dynamic Access Control Policies

Access control policies may depend on the runtime context of a collaboration environment. The privileges assigned to a user in a role may change with time due to the actions executed by other participants. Sometimes permissions may change due to the user's own actions, such as making a final agreement on a document, after which the creator of the document may not have the right to modify it. In some cases only during a specified period of time, certain operations may be performed. History-based access control policies, like various "separation of duties" constraints [22, 25, 21, 25] also require support for dynamic access control mechanisms. We use role admission constraints, and operation preconditions to facilitate the specification of such dynamic policies.

### 2.5 Hierarchical Organization of Activities

In an organization, many activities exist and new activities are constantly created. Existing roles and new roles participate in these activities. A large collaborative environment may sometimes need to be structured hierarchically. For example, a system supporting a team-based design project running over several weeks or months may need to have a number of dynamically created synchronous shared 'whiteboard-like" activities of relatively shorter life-spans. Such requirements for role based policy specification have been addressed using *domain* [17], *scope* [10], *role template* [9], and *authorization template* [14].

### 2.6 Decentralized Trust and Ownership Management

In a distributed collaboration, all users' nodes may not be equally trusted to maintain shared objects or enforce coordination and security policies. Policies specified on activities, roles and, objects need to be enforced at trusted sites. In a collaboration environment, activities may have participants from different administrative domains. In such a case, a single domain may not be trusted to manage all the activities and enforce collaboration policies. There needs to be a meta-role, such as the *Owner*, which is trusted by the participants to manage the activity. The selection of owners can be based on the fact that the owners have a vested interest to maintain the activity consistently. However, it may not be possible to trust the activity owner for managing all the roles within its scope, as participants in these roles may be from domains other than that of the owner. In such cases, different roles within the same activity may need to be managed by different *Owners* from different domains.

## 3 REQUIREMENTS FOR SECURE RUNTIME COLLABORATION SYSTEMS

The runtime environment, based on our collaboration model, requires management entities for objects, roles, and activities. A *role manager* needs to enforce policies for role admissions and operation invocations, and provide synchronization among collaborating participants. An *object manager* must enforce access control for shared objects. An *activity manager* is needed for securely distributing the collaboration policies to the role managers and object managers within its scope. Challenges in securely managing the runtime collaboration environment are discussed below.

### 3.1 User-Role Interaction

To perform any task within the runtime collaboration system, a user must join some role in the collaboration activity. The following issues need to be addressed during any interaction between a user and a role manager.

- During admission to a role, the role manager must authenticate the user and admit the user only if the admission constraints are satisfied.
- During an operation invocation, the role manager must authenticate the user, ensure that the user is a member of that role and check the operation preconditions before performing any operation related tasks.

### 3.2 User-Role-Object Interaction

In our model, coordination among collaborating users is enforced by operation preconditions and the role managers are responsible for maintaining the collaboration state. This requires that the users initiate their interactions with shared objects through role managers. If a role operation involves method invocations, the role manager is responsible for contacting the appropriate object manager. Once the object manager verifies the role operation context, it may provide the user a session with an interface through which the user may invoke objects' methods directly.

- Since the role manager invokes operations on behalf of the user, the role manager may need a ticket from the user which is presented to the appropriate object manager.
- As a malicious role manager may try to get extended privileges by not enforcing operation preconditions, the object managers may not always trust the role manager to enforce them. Since preconditions are a

means for enforcing dynamic access control policies, the object manager needs to verify the preconditions for the operations in whose context the method is being invoked.
- If an object's access control policy is based on users' identities or dynamic "separation of duty" policies, the role manager may need to communicate the user's identity to the object manager.
- If a session is setup between the user and the object, each subsequent method invocation by the user must have the appropriate session information for the object manager to perform access control.

### 3.3 Role-Role Interaction

Role managers are responsible for maintaining membership information.

- Role membership may depend on membership in another role. In such cases, the two role managers need to cooperate to enforce role constraints. Moreover, role membership information may need to be protected based on confidentiality and privacy requirements.
- Two roles can have operation preconditions that depend on each other, for example mutually exclusive operations. If these operations result in a common method invocation, the object manager allows only one request to proceed. However, if the operations do not have any such method invocation, a trusted entity needs to be assigned responsibility to enforce preconditions for such operations.

### 3.4 Secure Event Delivery

The requirements for secure event delivery are the following:

- A policy specifying which entities can generate specific events may be needed. Moreover, the source of an event needs to be verified and checked against this policy.
- Events need to be delivered only to authorized entities. In some cases, event filtering, where certain attributes of the event are hidden, may be required to prevent unwanted flow of information within the system.

### 3.5 Activity Creation

When a new activity is instantiated, new roles in the context of the new activity are created within the collaboration system.

- Activities are created as part of some role operation. Access control policies are required to specify who can create an activity. Instantiation of a nested activity within an existing activity must also be access controlled.
- Sites must be selected to manage the newly created activities as well as nested entities. In our model, the owner of a role or an activity is trusted to manage the corresponding entity. However, an *Owner* role can have multiple members, possibly spanning different administrative domains. Therefore, a mechanism is needed to select a site trusted by all the owners.
- The newly created roles may require access privileges for existing objects. The access control policies of these objects must be appropriately updated to reflect the existence of these new roles.
- In addition, policies for event subscriptions and notifications and access control need to be distributed to the managing sites.

## 4 OWNERSHIP MANAGEMENT OF ACTIVITIES, ROLES, AND OBJECTS

We present here an example to illustrate owner assignments to roles, activities, and objects in our collaboration model. Owner assignment rules are presented based on a trust relationship among the collaboration entities. The runtime system follows these rules to assign *trusted servers* to manage activities, roles, and objects. Additionally, we use the example to illustrate *activity template*, nested and concurrent activities, and object passing to nested activities.

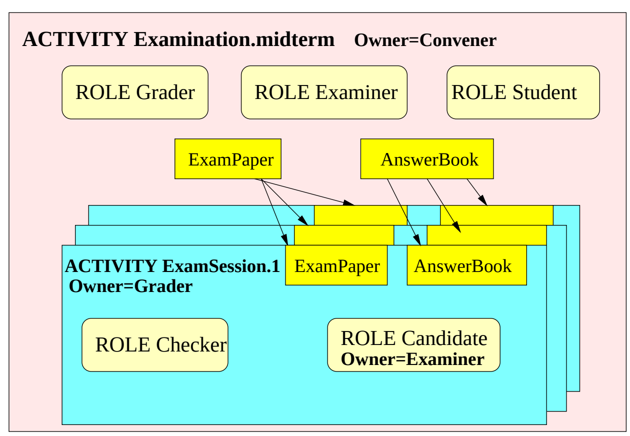


Figure 3: Hierarchical structuring of collaborative activities

In our model, an activity is defined using an *activity template* which encapsulates roles and object types. An *activity template* specifies a generic collaboration pattern among a set of roles using some shared objects.

Activities are instantiated from templates. Any number of instances of a template can be dynamically and concurrently created. Figure 3 shows an example of an *Examination* activity instance, named *midterm*. This examination activity contains three roles: *Grader*, *Examiner*, and *Student*. This *midterm* activity instance is created based on the *Examination* activity template, which has the nested *ExamSession* activity template. Each student creates an instance of the *ExamSession* activity to take the exam. An exam-session activity contains two roles: *Candidate* and *Checker*. The *midterm* activity instance has multiple concurrent exam-session activity instances initiated by different participants of the *Student* role. Only the student creating an exam session activity can be admitted to the *Candidate* role. A member of the *Grader* role is assigned to the *Checker* role for grading that exam session.

When an activity is instantiated, the user initiating that activity becomes a member of the *Creator* role for that instance. The creator may not always be trusted for managing the activity or a role within its scope. The *Owner* role, therefore, specifies which users can manage the activity instance or the roles. The example presented here is motivated to express the following requirements:

1. The student cannot be trusted to manage the exam session activity instance. Therefore, the *Grader* role of the *midterm* activity is made responsible for managing all the nested *ExamSession* activity instances.

2. Anonymity of the candidate during grading requires that the *Checker* role should not know the identity of the member in the *Candidate* role. Because the *Checker* role member is also a member of the *Grader* role, the *Grader* role cannot be trusted with the management of the *Candidate* role. Therefore, we specify that the management of the *Candidate* role to be done by the *Examiner* role.

3. References to two objects – *ExamPaper* and *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 exam session. The entity responsible for managing the *midterm* activity instance is responsible for managing all these objects.

In our model, an *Owner* role associated with an entity – an activity, role, or object – is trusted and delegated with the management tasks of that entity. The rules for assigning owners to entities at runtime are discussed below.

**Activity Owner Assignment Rules:**

1. The activity definition, i.e. *activity template*, may indicate which role is the owner of an instance of that activity template. The role assigned as the owner should be selected from the outer scope of the activity in our nested activity template structure. This ensures that when an activity instance is created, the role assigned as the owner is already created in its parent or ancestor activity instances.

2. If owner is not explicitly specified in an activity template, the owner of the parent activity becomes the owner of the new child activity.

3. In the nested activity structure, a *Convener* role is defined as the owner of the top level activity instance.

**Role Owner Assignment Rules:**

1. Similar to the first rule for activity owner assignment, the *activity template* may specify a role in the outer scope as the owner of a role.

2. If the owner for a role is not specified in an activity template, the owner of the activity instance is assigned as the owner of the role.

**Object Owner Assignment Rules:**

1. In our model, the owner of the role that creates an object becomes the owner of the object.

2. When an object is passed to a nested activity, additional access rights to the object, which are required by the roles in the nested activity, are added to the object's access control policy. Only if explicitly specified in the activity template, ownership for an object is passed to the roles of the nested activity.

In Figure 3, the *Convener* role is the owner of the top level *midterm* activity. By the default rule, the *Convener* is the owner of the *Grader*, *Examiner*, and *Student* roles. As expressed in requirement (1), the *Student* role is not trusted to manage the *ExamSession* activity instances. Therefore, the *Grader* role is specified as the owner for the nested *ExamSession* activity instances. As no owner is specified for the *Checker* role, by the default rule the *Grader* role, the owner of *ExamSession* activity instance, becomes the owner. Based on requirement (2), the *Examiner* is specified as the owner of the *Candidate* role. This ensures only the *Examiner* has access to the *Candidate* role's membership information and the participant's identity.

In the specification of this example, the *Examiner* role creates the *ExamPaper* object. Moreover, the *Student* role creates an *AnswerBook* object and initiates an *ExamSession* activity instance passing the *AnswerBook* object to the activity instance. As the *Convener* role is the owner for both the *Examiner* and *Student* roles, it is also the owner for the objects created by them, such as the *ExamPaper* and the *AnswerBook* objects. This satisfies the requirement (3).

**Trusted Server Selection Rules:** In our model, each role has a *trusted server*, which is trusted to manage entities owned by the role. These entities include activities, roles, and objects. A *trusted server* for a role is not the server where the role itself is managed, rather where the role manages the entities that it owns and is responsible to manage. In the above example, the trusted server of the *Convener* role manages the *midterm* activity instance and its roles. The trusted server of the *Grader* role manages all *ExamSession* activity instances and their nested *Checker* roles. On the other hand, the *Candidate* roles for all the *ExamSession* activity instances are maintained in the trusted server of the *Examiner* role. The trusted server for a role is assigned when an activity is instantiated.

## 5 MIDDLEWARE SERVICES

The runtime environment constructed for a collaboration by our middleware is composed of *activity managers*, *role managers*, and *object managers*. These components are organized in a tree structure, reflecting the static nesting of these entities in the specification. Each such node functions as the manager for the corresponding entity. Every participant in an activity maintains in his/her local host environment a replica of the execution tree for that activity. UCIs provide to the user mechanisms for participating in a collaboration through the local view of the execution tree. For any node in a tree, a trusted server contains the manager node, whereas the other replicas in participants' UCIs contain proxy objects for the manager node. A proxy object contains the Java RMI stub for the manager. An operation invoked on a node in the local tree, results in an invocation on the manager node.

UCIs and trusted servers provide protocols for authenticated remote operations and installation of new nodes in the local execution tree when requested by other trusted servers. This facilitates creation of tree nodes for new activities, roles, and objects. As new activities are instantiated, the participants' tree views in their UCIs are updated.

We illustrate through an example, shown in Figure 4, the runtime management structures. In our model, the middleware initiates the top level activity termed *system*. When a user initially joins a collaboration environment, the execution tree in his/her UCI contains only one node, which represents the *system* activity. This activity has a *Convener* role, which can load new activity templates, and create activity instances. In this example, the *Convener* role creates an instance of the *Examination* activity called *midterm* at its trusted server, *S1*. By the default ownership rule, the *Convener* becomes the owner of this new activity and its nested role managers. In this activity, user A is assigned to the *Student* role, user B is assigned to the *Examiner* role, and user C is assigned to the *Grader* role. At this point, the UCIs of these three users have been updated to reflect the *midterm* activity in their execution trees.

The entities with dashed outlines in Figure 4 are created as part of a *create activity* request for *ExamSession.1* by user A in *Student* role. In step 1, this request is send to the *Student's* role manager, which is managed at the *Convener's* trusted server *S1*. In step 2, following the ownership specification as discussed in Section 4, the new managers for *ExamSession.1* activity instance and its *Checker* role are created in the *Grader* role's trusted server, *S3*. Also, the manager for the *Candidate* role of the *ExamSession.1* activity instance is created in the *Examiner* role's trusted server, *S2*. As user A initiated the *ExamSession.1* activity instance, user A is assigned to the *Candidate* role. Moreover, user C of the *Grader* role is assigned to the *Checker* role. In step 3, the UCIs of these two users are updated to reflect the new roles in the context of the new *ExamSession.1* activity.

In our example, as part of the role's *create* operation,

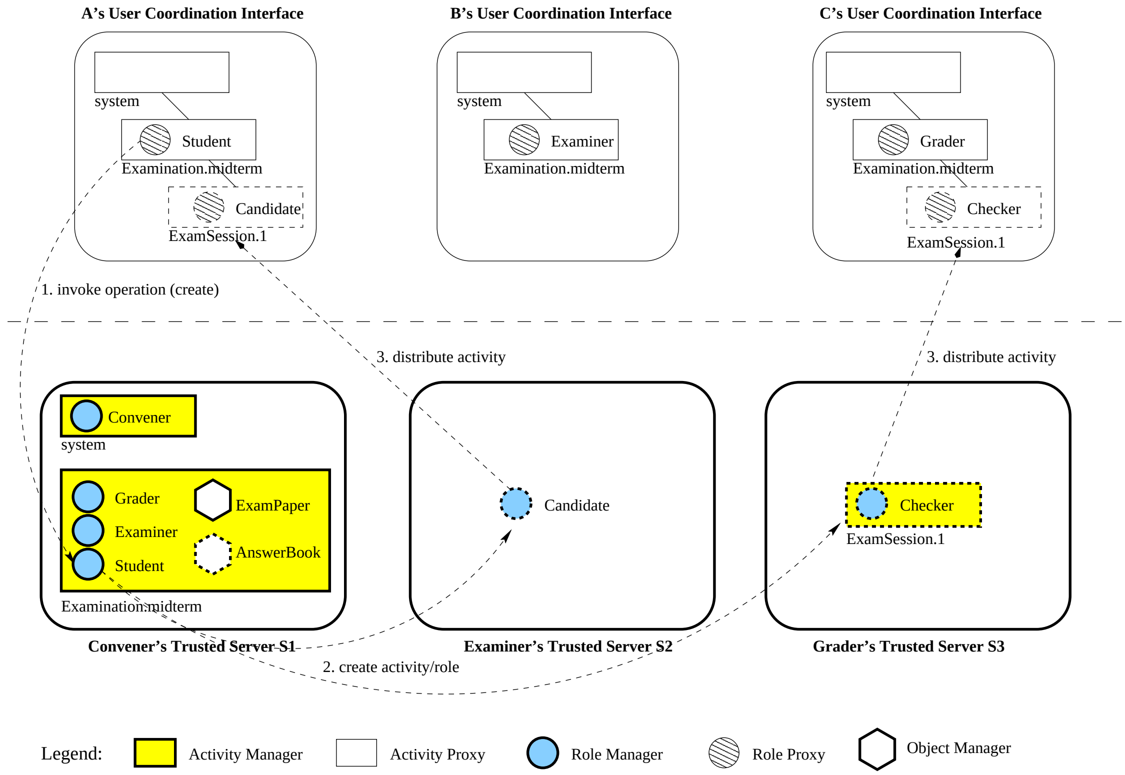


Figure 4: Activity creation and distribution – execution tree management

an *AnswerBook* object is created in the *midterm* activity and is passed to the new nested *ExamSession.1* activity instance. Also, the access control policies for the *AnswerBook* object and the existing *ExamPaper* object in the parent *midterm* activity are updated.

Following are the basic services provided by the middleware, which are required to securely manage the runtime environment.

### 5.1 Naming and Certificate Service

A secure name service assigns a globally unique name (URN) to every entity within a collaboration system. *Identity Certificates* are issued by the name service, representing a mapping between an entity's name and its public key. Names in our URN based name service are hierarchically structured. Hence, an instance of the *ExamSession* activity has the name:
*"system.Examination.midterm.ExamSession.1"*.
Role managers issue *Membership Certificates* to all role members. The owner of an activity issues *Ownership Certificates* to the owners of nested activities and roles. All certificates are verified by the issuer.

### 5.2 Secure Event Delivery

The middleware employs a peer-to-peer event service where event communication is based on authenticated remote invocations between publishers and subscribers. Activity managers, role managers, and object managers are the publishers and subscribers. Subscription and notification policies are derived from the collaboration specification and distributed to the appropriate managers at the time of activity creation. The *Subscription Policy* lists the events that the entity expects to receive from other entities, while the *Notification Policy* contains information about which other entities would be subscribing events from it. All incoming events are checked against the subscription policy to ensure their integrity. For auditing purposes each domain maintains all the events generated within its scope.

Role managers and object managers, if in different administrative domains, may not trust each other. In such cases, which entity - the role manager or the object manager - generates the *start* and *finish* events for an operation can have security consequences. When an operation does not involve any method invocations, as shown in Figure 2 case(i), the role manager generates the *start* and *finish* events. However, if two such role operations are inter-dependent, such as mutually exclusive operations, neither of the roles may be trusted to resolve the dependency as roles in our model can be competing. However, the collaboration entities are organized hierarchically in a tree structure following the ownership delegation rules presented in Section 4. We exploit the ownership hierarchy in our model to find an entity which can resolve such conflicts. In Figure 3, the *Checker* and *Candidate* roles in an *ExamSession* activity have different owners, *Grader* and *Examiner*. The *Convener*, being the owner of the *Grader* and *Examiner* roles, can resolve mutual exclusion like conflicts for the *Checker* and *Candidate* roles, if any.

On the other hand, when an operation has method invocations (Figure 2 cases (ii), (iii), (iv)), the role manager may not be trusted to generate the *start* event, as it may be to the role's advantage to send the event even when access is denied by the object manager. In our event service, the *start* event guarantees agreement between the role and object managers. In all these cases, the object manager is responsible for generating the *finish* event to signal the completion of the operation when the method invocation completes successfully (Figure 2 case (ii)), or when the user terminates the session (Figure 2 cases (iii) and (iv)). However to ensure that the role manager agrees that such an operation was indeed initiated, we need to ensure that the *finish* event is preceded by a corresponding *start* event. In our peer-to-peer event delivery model, the subscriber of the *finish* event also subscribes the corresponding *start* event.

## 6 PROTOCOLS FOR SECURE INTERACTIONS

We describe here protocols for secure interactions among the various entities in the collaboration.

### 6.1 Basic Authentication Protocol

The interaction protocols presented in this paper extend the basic challenge response protocol [18] shown in Figure 5. The tickets generated by these protocols are represented as

$$[signer's\ context,\ [message]_{Sign(signer)}]$$

An aggregated ticket is represented as

$$[Ticket_1,\ Ticket_2, ..]$$

The ticket, $Ticket_c$, generated at the end of the challenge response protocol is termed as the *basic ticket*, which is either aggregated with other tickets or extended for the secure interaction protocols presented here.

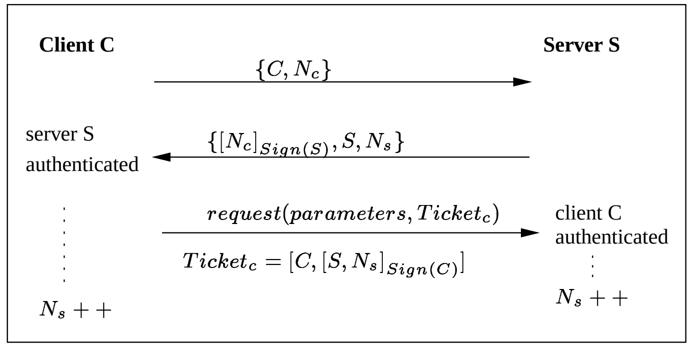


Figure 5: Challenge response protocol for authentication

1. Initially, the client sends its identity, and a challenge, $N_c$, to the server.

2. The server responds with its signature on the client's challenge to allow the client to authenticate the server, and provides the client with server's identity and a nonce, $N_s$, as a challenge to be signed by the client to authenticate itself.

3. The request from the client must contain a ticket, $Ticket_c$, which contains the client's identity, and it's signature on the nonce $N_s$. The server authenticates the client by verifying the client's signature on the nonce, $N_s$. The server's identity is included in the signature to thwart the "man in the middle attack". To prevent the client ticket $Ticket_c$ from being replayed, the nonce, $N_s$, is incremented in each subsequent request.

### 6.2 Role Operation Invocation Protocol

Figure 6 illustrates the *role operation invocation* protocol. It is executed after the execution of the basic authentication protocol. When an object is accessed by a user indirectly through a role, the role manager may need to present a ticket from the user to the appropriate object manager. This ticket is required to prevent the role manager from making unauthorized requests on behalf of a user. The basic ticket obtained in the challenge response protocol is aggregated with another ticket, which authorizes the role manager to invoke a method on behalf of the user in the given operation context. The protocol presented below corresponds to the message 1 in Figure 2.

To execute a role operation, a user invokes the generic *invokeOperation* method on the role manager providing a context-sensitive ticket, $Ticket_{op}$. The $Ticket_{op}$ contains the basic ticket, as well as a ticket, $Ticket_u$,

**User U** **Role R**

$$\xrightarrow{invokeOperation(opName, Ticket_{op})}$$

$$Ticket_{op} = [[U, [R, N_r]_{Sign(U)}], Ticket_u]$$
$$Ticket_u = [[R, u\_context]_{Sign(U)}, u\_context]$$
$$u\_context = \{U, UCI, opName, TS_u\}$$

Figure 6: Role operation invocation protocol

which authorizes the role to act on the user's behalf. $Ticket_u$ contains the user's signed context, which consists of his/her identity, *U*, the user's *UCI* information, the operation being invoked, and a timestamp, $TS_u$. The signature also contains the role's identity to prevent the role manager from delegating the authorization to another role manager. By including the timestamp in the context, replay of the user's context is prevented. Since the signed context includes the operation name, a malicious role is prevented from changing the operation context.

### 6.3 Role-Object Interaction Protocols

When the role manager invokes an object method, it may need to provide the user's context to the object manager for access control. The protocol for the specific case when the object manager needs the user's signed context is shown in Figure 7.

**Role R** **Object O**

$$\xrightarrow{objRef.method(parameters, Ticket_m)}$$

$$Ticket_m = [[R, [O, N_o]_{Sign(R)}], Ticket_u]$$
$$Ticket_u = ticket\ provided\ by\ user\ U$$

Figure 7: Object method invocation protocol

1. The role manager includes with the method invocation a ticket, $Ticket_m$, which contains a basic ticket signed by the role manager as well as the user's authorization ticket, $Ticket_u$ which was provided to the role manager during the *role operation invocation* protocol described earlier.

2. The object manager verifies the basic ticket, and ensures that the role manager's identity matches the identity provided in the user authorization ticket, $Ticket_u$. The user's identity from this ticket, $Ticket_u$, is then used for enforcing any user-id based access or dynamic "separation of duties" constraints.

### 6.4 User-Object Interaction Protocol

In our model, a user may have extended interactions with an object, which is initiated through a role operation, as shown in Figure 2 case(iv). To facilitate direct user-object interaction, a session is established by the object manager when requested by a role. During session establishment, message 4 in Figure 2 case(iv), the object manager needs to prove to the user that the user has requested this session through a role operation. For this specific case, when a session establishment is required, the protocol presented in Figure 7 is extended to include an additional ticket as shown in Figure 8. This interaction between the role and object corresponds to the message 2 in Figure 2 case(iv).

**Role R** **Object O**

$$\xrightarrow{objRef.method(parameters, Ticket_m)}$$

$$Ticket_m = [[R, [O, N_o]_{Sign(R)}], Ticket_{ro}]$$
$$Ticket_{ro} = [[O, Ticket_u]_{Sign(R)}]$$
$$Ticket_u = ticket\ provided\ by\ user\ U$$

Figure 8: Object method invocation protocol to initiate a user-object session

In Figure 8, an additional ticket, $Ticket_{ro}$ is provided to the object manager when a session needs to be initiated. Having verified the basic ticket, $[R, [O, N_o]_{Sign(R)}]$, from the role, the object manager initiates a session establishment protocol. The protocol for user-object session is shown in Figure 9.

**User U** **Object O**

$$\xleftarrow{\{interfaceRef, O, N_{ou}, session\_id, Ticket_{ro}\}}$$
$$Ticket_{ro} = Ticket\ provided\ by R\ to\ O\ to\ initiate\ a\ session$$

$$\xrightarrow{interfaceRef.method(paramaters, Ticket_s)}$$
$$Ticket_s = [[N_{ou}, O, s\_context]_{Sign(U)}, s\_context]$$
$$s\_context = \{U, UCI, session\_id, TS_{uo}\}$$

Figure 9: User-Object interaction protocol

1. The object manager initiates a challenge response protocol with the user by sending the object interface, the object manager's identity, a session id, a nonce, $N_{ou}$, as well as the ticket, $Ticket_{ro}$, provided by the role manager ( Figure 8). The $Ticket_{ro}$ contains $Ticket_u$, the role's authorization from the user, and identity of the object to whom the role has passed the authorization.

2. The user verifies that the user context provided in the ticket, $Ticket_{ro}$, is the same as that given by the user to the role manager when invoking the role operation. The object manager's identity in the ticket allows the user to verify that the interface is really coming from the right object manager. The object manager associates the session id with the user and the role operation context.

3. To invoke a method directly on the object, the user includes a session sensitive ticket, $Ticket_s$. This ticket contains the user's signature on the nonce, $N_{ou}$, and the user's context, which includes the user's identity, UCI location, the session id, and a timestamp, $TS_{uo}$. The nonce, $N_{ou}$, is incremented with each method invocation. The signature includes the session id to ensure the integrity. Replay of the ticket, $Ticket_s$, is prevented by including the timestamp, $TS_{uo}$, in the context.

4. During a session, one of the method invocations on the underlying object signifies the end of the session. On invocation of this method, the object manager invalidates the session id, nullifying the user's authenticated role context. The *finish* event is generated by the object manager to signify the completion of the role operation which initiated the user session.

The secure interaction protocols presented in the section are based on the interaction models presented in Section 2.1. Our requirement of initiating object interactions through roles, ensures the confidentiality of roles' internal information and various policies, such as activation constraints and intra-role coordination policies.

### 6.5 Activity Creation and Distribution Protocol

In our model activities are created either by installing a new *activity template* and instantiating it or by instantiating a nested activity through a role operation. The installation of an *activity template* and its subsequent instantiation is usually performed by the *Convener* on its trusted server. From the security point of view, this activity creation protocol is simple as all the steps in the protocol are performed on the *Convener's* trusted server. However, when a nested activity is instantiated, the activity creation and distribution protocol has to manage several security aspects:

- The role manager, which initiated the nested activity creation, contacts its activity manager to create the new activity instance. The activity manager, on behalf of the initiating role manager, contacts the owner of the nested activity to create the new activity instance's manager.
- The initiating role manager has to prove to the role manager of the new activity's owner that it is authorized to create the requested activity.
- Similarly, if the activity has encapsulated roles, the role managers of the owners of these roles have to be contacted and provided with the proof that the initiating role manager has the rights to create the corresponding role managers.
- The new activity and its nested roles have to be created on their owners' trusted servers.
- If users are assigned to roles as part of activity instantiation, the new role managers have to update the users' UCIs with the user specific views of the activity.

Using the example in Figure 4, here we present the steps for creating and distributing *ExamSession:1* activity instance. However, we only present the steps for *create role* request for the *Candidate* role initiated by the *Student* role manager to the *Examiner* role manager.

1. The owner of the *midterm* activity, *Convener*, delegates the ownership of the new *Candidate* role to the *Examiner* role with the following certificates: (Identity Certificates for the *Convener* role is not shown)
   - $[Owner(..Examiner, ..Candidate)]$: a role ownership certificate signed by the owner of the *midterm* activity, i.e., the *Convener*. This signifies the delegation of the ownership of the *Candidate* role. All names are hierarchically nested (however, not shown in this discussion for simplicity). For example, the fully qualified name of the *Examiner* is "system.Examination.midterm.Examiner".
     This nested naming ensures that the *Examiner* role in the certificate is indeed created in the scope of *midterm* activity instance.
   - $[Owner(Convener, ..midterm)]$: an activity ownership certificate signed by the *Convener* role. This proves that the *Convener* is the owner of the parent *midterm* activity and can delegate the ownership of the *Candidate* role.

2. The *Examiner* role manager verifies *Student* role's access right to create the *Candidate* role.

3. The *Examiner* role manager contacts its trusted server

*S2* to create the role manager for the *Candidate* role. The certificates provided with the *create role* request are shown in Figure 10. These certificates are required by the new *Candidate* role manager when delegating ownership of the entities it manages and owns.

**Role Manager Examiner** $\xrightarrow{create\ role}$ **Examiner's Truster Server S2**

$[Owner(..Examiner, ..Candidate)]_{Sign(Convener)}$
$[Owner(Convener, ..midterm)]_{Sign(Convener)}$

Figure 10: Certificates to create the *Candidate* role

4. The role manager for the *Candidate* role is created at the trusted server *S2*. This role manager initiates a *distribute activity* for its members, i.e., user A.

5. The user A's UCI authenticates the *Examiner* role manager using basic ticket of challenge-response protocol.

User A's UCI then inserts the proxy objects into it's execution tree. A's view of the activity contains the proxy for the *Candidate* role. During distribution, user A is also provided with its membership certificate for the *Candidate* role. Similarly the proxy for the *Checker* role in the context of the *ExamSession.1* activity is distributed to user C's UCI.

## 7 RELATED WORK

The existing research on security in collaboration systems targets either synchronous groupware applications or asynchronous workflow. However, the security requirements and solutions proposed for these two areas differ. In groupware environments, an access control model with unique sets of rights for shared-view based GUI interface control has been presented in [6]. In workflow, database oriented security is prevalent [2, 14]. A task-based constraint specification language for workflow management systems is discussed in [2]. There, constraints are specified with a mapping between roles and tasks. SecureFlow [14] imposes workflow authorization constraints on tasks using Authorization Template (AT), similar to *activity template* in our model. AT is a tuple specifying privileges to be granted to a subject of a given role on an object of a given type during a given time interval. In contrast, an activity template is a collaboration pattern, which involves multiple roles, objects, and their coordination. Decentralized management of workflow system to enforce Chinese wall security policies are presented in [1] introducing concepts, such as self-describing workflows and workflow stubs. Compared to these works, in our collaboration model, we are concerned with a role based specification of collaboration environments using activity templates and automated realization of such distributed activities.

Our goal is similar to those of COCA [15] and DCWPL [3] in their approach of constructing a distributed collaboration environment from a high level specification. COCA [15] is a logic-based coordination policy specification language for interactive CSCW applications. DCWPL [3] addresses user level mechanisms to deal with group interaction issues and is limited to its predefined policies and functions. Intermezzo [7] is one of the first systems to introduce the basic concepts of role based policies in CSCW systems, with primary focus on user-presence awareness environments. However, the concept of role has been used in the past in many CSCW systems[12, 4]. These systems do not address most of runtime security requirements presented in this paper, such as management of user interactions with roles and shared objects, and secure protocols for activity management.

The concept of *cascaded authentication* is presented in [24], in the context of delegating client requests from one server to another. The idea of delegating authority is extended in [16] using *self authenticating proxies* where authorization is embedded in a signed token. The tickets used in our secure interaction protocols serve similar purpose of *cascaded authentication*[24] and self authenticating proxies[16]. A distributed and decentralized management of roles in a distributed service model is presented in [13]. In comparison, we take into consideration the trust among the collaboration entities for delegating management responsibilities.

## 8 CONCLUSIONS

The focus of this paper is on securely managing roles, shared objects, activities, and user interactions in a role based distributed collaboration. We have presented here security requirements of a policy driven collaboration system and its runtime environment for managing interactions among users, roles, activities, and objects. Rules for managing ownership of the distributed collaboration entities, such as activities, roles, and objects, are presented in this paper. Importantly, based on different interaction models, protocols for secure role operation invocation, user-role-object interactions, activity and role creation and distribution are presented. These protocols are based on distributed trust relationships among various entities, considering the fact that all participants and sites in a distributed collaboration system may not be equally trusted. The protocols presented here utilize a ticket based mechanism to delegate the appropriate management tasks to different sites based on a trust relationship model.

## REFERENCES


1. Vijayalakshmi Atluri, Soon Ae Chun, and Pietro Mazzoleni. A Chinese wall security model for decentralized workflow systems . In *Proceedings of the 8th ACM Conference on Computer and Communications Security* , pages 48 – 57, 2001.

2. E. Bertino, E. Ferrari, and V. Atluri. A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems. In *ACM Workshop on Role-based Access Control*, pages 1–12, 1997.

3. Mauricio Corts and Prateek Mishra. DCWPL: a programming language for describing collaborative work. In *Proc. of CSCW'96*, pages 21 – 29, November 1996.

4. S.A. Demurjian, T.C. Ting, and B. Thuraisingham. User-role based security for collaborative computing environments. *Multimedia Review*, 4(2):40–47, Summer 1993.

5. Yvo Desmedt. Society and group oriented cryptography: A new concept. In *Advances in Cryptology – Proceedings of CRYPTO '87*, pages 120–127. Springer, LNCS #293, 1988.

6. Prasun Dewan and HongHai Shen. Controlling access in multiuser interfaces. *ACM Transaction Computer-Human Interaction*, 5(1):34 – 62, March 1998.

7. W. Keith Edwards. Policies and Roles in Collaborative Applications. In *Proc. of CSCW'96*, pages 11–20, 1996.

8. Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: some issues and experiences. *Communication of ACM*, 34(1):39 – 58, January 1991.

9. Luigi Giuri and Pietro Iglio. Role templates for content-based access control . In *Proceedings of the second ACM workshop on Role-based access control* , pages 153 – 159, November 1997.

10. Cheh Goh and Adrian Baldwin. Towards a more complete model of role. In *ACM Workshop on Role-based Access Control*, pages 55 – 62, 1998.

11. S. Greenberg. Personalizable Groupware: Accomodating Individual Roles and Group Differences. In *In Proceedings of the European Conference on Computer-Supported Cooperative Work(ECSCW)*, pages 17–31, 1991.

12. Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Information Systems*, 5(2):187–211, 1987.

13. 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.

14. Wei-Kuang Huang and Vijayalakshmi Atluri. SecureFlow: a secure Web-enabled workflow management system. In *ACM Workshop on Role-based Access Control*, pages 83 – 94, 1999.

15. Du Li and Richard Muntz. COCA: Collaborative Objects Coordination Architecture. In *Proc. of CSCW'98*, pages 179–188, 1998.

16. Marie Rose Low and Bruce Christianson. Self Authenticating Proxies. *The Computer Journal*, 37(5):422–428, 1994.

17. Emil C. Lupu and Morris Sloman. Reconciling Role-Based Management and Role-Based Access Control. In *ACM workshop on Role-based Access Control*, pages 135–141, 1997.

18. Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

19. P. Roberts and J-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proc. of IFIP Congress*, pages 981–986, 1977.

20. Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-Based Access Control

Models. *IEEE Computer*, 29(2):38–47, February 1996.

21. Ravi S. Sandhu. Transaction control expressions for separation of duties. In *Fourth Annual Computer Security Application Conference*, pages 282–286, December 1988.

22. R.T. Simon and M.E. Zurko. Separation of duty in role-based environments. In *10th Computer Security Foundations Workshop*, pages 183 –194, 1997.

23. Suzanne Sluizer and Paul M. Cashman. XCP: an experimental tool for managing cooperative activity. In *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, pages 251 – 258, 1985.

24. Karen R. Sollins. Cascaded Authentication. In *IEEE Symposium on Security and Privacy*, pages 156–163, 1988.

25. Jonathon E. Tidswell and Trent Jaeger. Integrated constraints and inheritance in DTAC. In *ACM Workshop on Role-based Access Control*, pages 93 – 102, July 2000.