A Decentralized Access Control Model for Dynamic Collaboration of Autonomous Peers

Stefan Craß(✉), Gerson Joskowicz, and Eva Kühn

Institute of Computer Languages, TU Wien, Argentinierstr. 8, Vienna, Austria
{sc,gj,ek}@complang.tuwien.ac.at

Abstract. Distributed applications are often composed of autonomous components that are controlled by different stakeholders. Authorization in such a scenario has to be enforced in a decentralized way so that administrators retain control over their respective resources. In this paper, we define a flexible access control model for a data-driven coordination middleware that abstracts the collaboration of autonomous peers. It supports the definition of fine-grained policies that depend on authenticated subject attributes, content properties and context data. To enable peers to act on behalf of others, chained delegation is supported and permissions depend on trust assumptions about nodes along this chain. Besides access to data, also service invocations, dynamic behavior changes and policy updates can be authorized in a unified way. We show how this access control model can be integrated into a secure middleware architecture and provide example policies for simple coordination patterns.

Keywords: ABAC · Delegation · P2P · Coordination middleware

1 Introduction

Modern distributed systems are often not managed by a single organization, but require collaboration of multiple stakeholders that provide data and offer services. Due to evolving application requirements and availability of different providers for specific tasks, distributed workflows should be dynamically configurable and enable ad-hoc coordination. Examples for such complex interactions include cloud-based business-to-business transactions, peer-to-peer (P2P) networks that enable efficient data replication, and connected smart devices.

As mutual trust cannot be assumed in such dynamic communication networks, a suitable access control model is necessary that enables participants to specify who can access their data and services. To address the flexibility of distributed systems with dynamically changing security requirements, each member shall be able to manage its own access control policy independently of others [1]. This requires an authentication concept that supports identity providers from different security domains, which may be linked to different trust levels. In order to cope with indirect access on behalf of other users, support for delegated identities is needed. For instance, a customer may want to access a company's data

B. Thuraisingham et al. (Eds.): SecureComm 2015, LNICST 164, pp. 519–537, 2015.
DOI: 10.1007/978-3-319-28865-9_28

storage via a cloud service. The company may allow a trusted cloud service to read data associated with the delegating customer, while denying direct customer access in order to make security administration simpler and more reliable.

In order to adhere to the principle of least privilege, permissions shall be specifiable in a fine-grained way. Access decisions may depend on the environmental context (e.g. previous interactions), while the administration of policies itself shall be governed with meta-level policies [2]. For instance, resource owners may delegate their administrator privileges to other trusted users, or a cloud provider may allow users to control access to their deployed services themselves.

Current security mechanisms for distributed systems usually rely on centralized servers, which limits their use to networks controlled by a unified administration. There still is a lack of powerful security models for the collaboration of autonomous peers in dynamic scenarios. Although some research has been done on decentralized authentication and authorization [3,4,5], most approaches do not model fine-grained access control policies that support content- and context-based rules as well as arbitrary forms of delegation.

In this paper, we present a flexible and expressive security concept that targets the dynamic coordination of autonomous components in a fully decentralized environment. We assume that applications are designed using a data-driven coordination model [6], which hides the complexity of remote communication and provides intelligible abstractions for service invocation and data access. Application logic is encapsulated in decoupled software components termed *peers*, whose interactions are specified declaratively. Although the security mechanisms are shown in the context of this specific architecture, the concept is applicable to any business process that is implemented using interconnected components.

We propose an extended middleware architecture for this coordination model called *Secure Peer Space* that enables decentralized authorization with support for complex delegation chains and fine-grained access control rules. Rules depend on the accessed content, the environmental context and the subject. We combine elements of attribute-based and discretionary access control, as decisions are based on authenticated attributes, while each owner of a peer may govern access to its own services and data. In contrast to usual access control concepts that place controls on few entry points, we support access control at any involved component of a workflow. The access control model is suitable for cross-organizational collaboration, as it provides a way to specify trust in attributes from distributed sources. It is also possible to depict multitenant scenarios, as users may dynamically inject sub-peers into another peer if permitted by its owner. The security mechanisms, including policy administration, are largely bootstrapped using existing coordination features of the middleware.

The paper is structured as follows: Section 2 describes the addressed coordination middleware. On top of that, Section 3 presents a security concept and a middleware architecture for the Secure Peer Space. Section 4 provides examples for the usage of this secure middleware in the form of reusable coordination patterns. Section 5 discusses the benefits of the presented approach and compares it to related work. Finally, Section 6 concludes the paper and outlines future work.

2 Modeling Coordination with the Peer Model

When designing distributed applications, middleware can help to hide the complexity of remote communication and offer proven coordination primitives for common tasks like synchronization, data access and service invocations. A coordination model provides a high-level abstraction on how to program the coordination logic of a distributed system, i.e. the interactions of individual software components. The *Peer Model*, originally described in [6], allows for modeling of data-driven workflows among highly decoupled components (i.e. peers) in a distributed environment. In the following, we describe the basic concepts of the Peer Model and its associated middleware runtime, the *Peer Space*.

A peer is an addressable component consisting of *space containers* [7], which hold its internal state, and *wirings*, which connect containers within and between peers. Thus, wirings describe the component behavior and its coordination logic. Space containers, which are inspired by Linda tuple spaces [8], store typed data items termed *entries* and provide methods to write and query them. Different kinds of data and messages that are required in a distributed system can be modeled by entries, including events, user data, service requests and responses. Besides its payload, each entry contains a set of (possibly nested) key/value pairs termed *coordination properties*, which determine how an entry is affected by wirings. Each peer provides a *Peer-In-Container* (**PIC**) and a *Peer-Out-Container* (**POC**), forming its input and output stages, respectively. Peers may also be nested so that parts of their functionality are encapsulated into sub-peers.

Wirings are triggered by a specific combination of entries, execute application-specific logic and output their results as entries. A wiring consists of one or more *guards*, zero or more *services* and zero or more *actions*. Guards impose certain conditions on the content of containers. When all guards of a wiring are fulfilled, they provide the services with a set of input entries from these containers. The services may modify the entries or create new ones based on the input. The resulting output entries are then distributed to their target containers by the actions. A wiring may only access containers of the enclosing peer and those of its direct sub-peers. Each guard is specified via a source container, an operation type, and a query that selects a certain subset of entries in the given container. By default matching entries are deleted from the source container when a wiring is triggered (operation type "**move**"), but they may also be only read (operation type "**copy**"). The query consists of the required entry type, optional selectors on further coordination properties (e.g. "$[size < 10\ kB]$"), and a count parameter, which defines the minimum and maximum number of entries to be selected (default: $min = max = 1$). A query is only fulfilled when at least the minimum number of entries matching the type and selector criteria are available in the source container. After the services have been executed, the actions operate on the resulting entries. Each action has a query that selects from these entries and a target container where matching entries are written to. Unlike guards, an action does never block as the output entries are fixed after service execution.

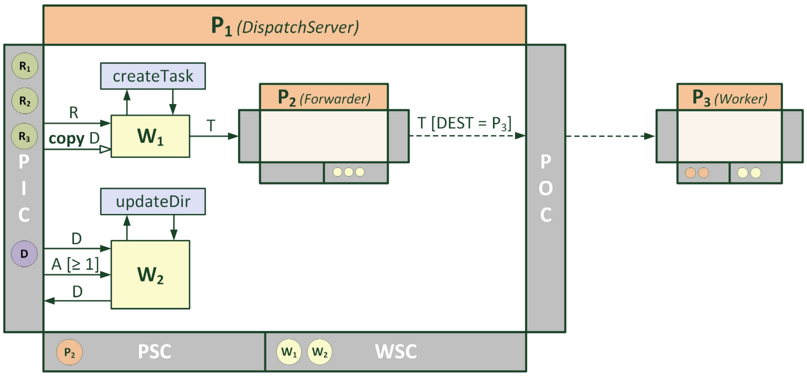

Fig. 1. Peer Model example with dynamic state.

The Peer Space middleware runtime executes the modeled wirings and realizes remote communication between peers. We denote a specific instance of a collaborative interaction as a *flow*, which is usually started by a single user request and may involve the (possibly concurrent) execution of several wirings located at multiple (possibly distributed) peers. To provide the glue that creates a distributed process out of the modeled behaviors of involved peers, there are several predefined coordination properties that determine how entries are treated: A unique **FlowID** helps the framework to correlate entries that belong to the same flow. Timing constraints may be addressed by time-to-live (**TTL**) and time-to-start (**TTS**) properties, which limit the lifecycle of an entry and delay its activation, respectively. The destination (**DEST**) property of an entry provides the mechanism to model directed remote communication. It specifies the target container using the address of its peer and a container name (default: *PIC*). The entry is then injected into this container by the Peer Space.

We also introduce a meta-model approach for the dynamic adaptation of behavior by adding and replacing wirings and peers. Besides PIC and POC, each peer also has a *Wiring Specification Container* (**WSC**) and a *Peer Specification Container* (**PSC**). Each wiring corresponds to a meta entry in the WSC that includes the wiring specification as payload. Similarly, the PSC contains the specification entries for each direct sub-peer. The behavior of the sub-peers is then specified recursively. These meta entries may be accessed like regular entries in a PIC or POC. Thus, they can be injected via remote communication, may be written or deleted by local wirings and are garbage-collected based on their TTL. This mechanism is also required to allow queries with parameters that dynamically depend on the application logic. For that, a wiring must create a suitable wiring specification entry in its service and write it to the WSC of the corresponding peer. Depending on its specification, such a dynamic wiring may run as continuous subscription until explicitly deleted or only as a one-off query.

Fig. 1 outlines an example model that dispatches tasks to remote worker peers based on client requests and a configurable lookup directory. Wiring W_1

takes one request (of type R) and a copy of its internal lookup directory D from the PIC of peer P_1 and passes them to its service, which creates a task of type T for a specific worker peer. The wiring's action writes this entry to the PIC of sub-peer P_2, which is responsible for reliably forwarding the task to the target peer. Its sending action is indicated by the dashed arrows, which can be viewed as wirings that are dynamically set by the runtime when it encounters an entry with a specified destination, like P_3 in our example. To keep the model simple, the internal behavior of P_2 and P_3 is not detailed here. The second wiring W_2 updates the lookup directory by taking the corresponding entry together with any new advertisements of type A that have been sent to the peer to indicate changes in the list of available peers. The example also shows the dynamic state of the model during execution. We assume that there are currently three requests and one directory entry in the PIC of P_1. This means that W_1 can be triggered three times, while W_2 is currently waiting for at least one entry of type A. Finally, the figure also depicts the meta model, as sub-peers and wirings are represented by corresponding entries in the PSC and WSC, respectively.

3 Security for the Peer Model

The Peer Model supports a flexible and comprehensible way of modeling coordination within distributed applications, but it lacks a suitable security model. In the following, we describe Peer Model extensions that provide the required security concepts and a corresponding middleware architecture for the Secure Peer Space. The main elements are an attribute-based representation of identities with support for chained delegation, a fine-grained rule-based authorization mechanism for access to entries in regular containers and the meta model, and a secure runtime architecture that authenticates incoming entries and enforces access control on them. The proposed security concept is based on previous work on an access control model for space containers [9], which is adapted to the needs of the Peer Model. Major additions are a trust model based on delegation chains, support for dynamic behavior changes via meta containers, and the introduction of hierarchic policies based on nested peers.

3.1 Identity Representation with Delegation Support

A unified representation of identities forms the foundation of the decentralized security model. We use the notion of a *principal*, which represents a specific user (i.e. a system or a person) within the distributed system. The identity of a principal is represented by a data set managed by an *identity provider*. For each runtime, there is an explicit principal termed *runtime user* that represents the Peer Space when communicating with remote runtimes.

Management of permissions must be scalable. Instead of assigning permissions to each user separately, access control should rely on roles and other attributes of authenticated principals. Therefore, the Secure Peer Space supports attribute-based access control (ABAC) [10], where rules depend on one or more validated attributes. Role-based access control (RBAC) [11], where rules grant access for principals with specific roles, can be seen as a special case of ABAC, as role information can be included via attributes. Additional attributes may vary, but at least a user ID and an associated domain (e.g. the user's organization) must be included to be able to uniquely identify the principal.

Access control in the Secure Peer Space targets any operation on containers, which includes (possibly consuming) queries by wiring guards, write operations by wiring actions and entry injection via remote communication. The responsible entity for a specific operation with regard to access control is called the *subject*. A subject may correspond to a single principal, but it may also represent a composition of several principals in case of delegation. The subject that writes an entry to a container is assigned as the *entry owner*. Its identity is represented via nested *subject properties*, which are a special form of coordination properties that are attached to each entry and represent the authenticated attributes.

As peers and wirings are specified by writing entries into meta containers, they also have dedicated owners, which are called *peer owners* and *wiring owners*, respectively. Peer owners are able to administrate the access control policies of their peers, which is detailed in Section 3.2. Wirings and sub-peers are usually created by the owner of their parent peer, but they may also be inserted dynamically by different subjects that were authorized by the peer owner. Whenever a wiring tries to select entries from a container via its guards, the wiring owner is the relevant subject for checking if the wiring is allowed to do so. Similarly, when entries need to be written by a wiring action, the corresponding wiring owner determines the subject relevant for access control and thus also the owner of the emitted entries. The entry owner is not necessarily the original creator of an entry. Even if a wiring modifies only some properties of an entry or simply forwards it to another container, the entry owner is changed. A wiring may choose to use *direct access* and set the entry owner to the wiring owner, or it may apply *indirect access* on behalf of another subject to support delegation.

The simplest way to model delegation would be for a server to use the provided credentials from a user to authenticate at another site and perform some action on the user's behalf. However, this would allow servers to impersonate other principals, which is not feasible as we do not assume inherent trust in any principal. The path to the ultimate target of a request from the initial request issuer may involve several machines that are not equally trusted [12]. A suitable delegation concept for the Secure Peer Space must therefore support chained delegations (e.g. $User1$ delegates to $User2$, who delegates to $User3$ etc.) and allow access control decisions that depend on a combination of the involved principals. The first element of the delegation chain is the initial issuer of a request, while the last element corresponds to the principal that has actually written the entry to its current container. For better readability, we depict a subject by listing the principals in reverse order ($User3$ **for** $User2$ **for** $User1$). For most examples in this paper, we just specify the principals' user IDs, even though their identities actually consist of multiple attributes that are stored in the subject properties.

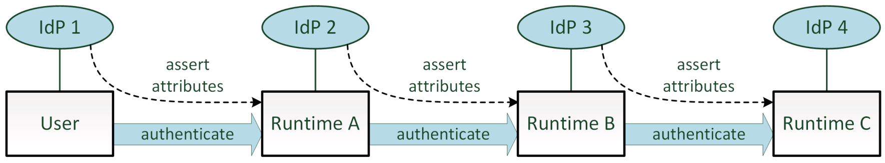

Fig. 2. Chained authentication with different identity providers.

Delegation is triggered when a service within a wiring emits an entry using indirect access mode. Subject properties of an entry cannot be directly manipulated by the service, which prevents malicious wirings from issuing requests or writing data on behalf of arbitrary principals. Instead, they may select any of the owners of their input entries as delegating subject. Thus, a wiring with owner A that fetches two entries with owners B and C, respectively, may use the subjects "A **for** B", "A **for** C", or simply "A" (when using direct access). When another wiring (with owner D) subsequently copies or moves this new entry, it may itself use indirect access and add its own identity to the delegation chain (e.g. "D **for** A **for** B"). This approach prevents impersonation, as the wiring owner is always included as the responsible actor for any action performed by the wiring. Thus, delegation may be used to restrict access based on the identity of the delegating subject, but not to escalate the privileges of the wiring owner, which still has to be authorized to induce the subsequent steps in the flow.

As authentication is performed in a decentralized way, additional challenges arise. Principals that are part of a subject need not necessarily be directly authenticated at the local Peer Space. For instance, when a delegation chain spans several runtimes with runtime users U_1 to U_n, the last Peer Space only directly authenticates U_{n-1} and it has to trust this runtime that U_{n-2} has indeed been correctly authenticated and so forth. A similar problem occurs when a wiring with an owner different from the local runtime user wants to send an entry to a remote Peer Space (using the DEST property). As the runtime must not impersonate the wiring owner directly, it authenticates at the remote site using the identity of its own runtime user, while claiming that the wiring owner has been authenticated at its site (or at another runtime that it trusts).

Each principal in a delegation chain may be authenticated by a different Peer Space runtime, which is illustrated in Fig. 2. In this example, a separate entity (IdP 1-4) is responsible for asserting attributes for each principal, but multiple runtimes may also share the same identity provider. The identity providers act as anchors of trust, which prove the validity of the authenticating principal's identity using some form of authentication mechanism (e.g. certificates) not detailed in this paper. However, they are not responsible for the identity of any previously authenticated principal in the chain. Instead, the runtime that has authenticated a principal has to guarantee the validity of the claimed subject properties when it forwards this information. It is not only necessary to trust that the principals in the delegation chain are in fact acting legitimately on behalf of previous principals, but also that the claimed identities of these principals are in fact valid and were not modified by any runtime on the path from the original authenticator to the current runtime. Therefore, each principal in the delegation chain is associated with a separate authentication chain that specifies the identities of the runtimes that have forwarded its authenticated attributes.

The delegation concept is based on the establishment of explicit trust relationships among collaborating principals by means of access control rules. Each peer owner can independently select which forms of delegations are trusted by including constraints on the delegation and authentication chains of an incoming entry. For instance, it may be specified that only peers with a certain role may act on behalf of other principals and that owners of the delegating peers must be authenticated by a runtime owned by a specific organization. If such rules are defined for every peer, an explicit chain of trust can be established that states which peers and runtimes are trusted to act on behalf of prior peers.

As the entries that constitute a delegated access and their authentication data are relayed on the same path (via the participating Peer Spaces), the delegation chain and the individual authentication chains can be combined in a single data structure called *subject tree*. The root of this tree represents the local runtime user. Its direct children are the principals within the subject that were directly authenticated by the runtime. Each inner node depicts a runtime user that has been authenticated by its parent and has authenticated its children. The leaves correspond to principals that are part of the delegation chain, while the path from each leaf to the root forms the respective authentication chain. The order of principals in the delegation chain is defined via a left-to-right tree traversal.

This subject tree is iteratively extended as entries are processed and forwarded along a flow. When an entry is received from a remote runtime, the authenticated security attributes of the sender are written to the root node of the entry's subject tree. Then, the local runtime user is added as the new root. When a service triggers delegation using indirect access, the subject trees of the delegating input entry owner and the wiring owner are merged to form the subject tree for the output entries. As both root nodes represent the local runtime user, they are replaced by a common root, whereas the child nodes associated with the wiring owner are placed after those of the input entry owner to ensure the correct order of the delegation chain.

An example for such a subject tree is shown in Fig. 3, which can be mapped to the Peer Model example from Fig. 1. The subject tree depicts the owner of the task entry that arrives at peer P_3, assuming that the original request came from another remote peer P_0 owned by user "evakuehn" from TU Wien and that P_1 together with its sub-peer and wirings are owned by a system user from organization "OrgA". Each of the three peers P_0, P_1 and P_3 are hosted by different Peer Space runtimes with separate runtime users. The delegation chain can be represented as "$SystemUser$ **for** $evakuehn$", whereas the other principals are part of the authentication chains. User "evakuehn" has been authenticated by runtime user "SBCServer" when she has registered P_0. When sending the request from P_0, this runtime then authenticates at the runtime of P_1 (with runtime user "Server123"), which happens to be a cloud node that may host

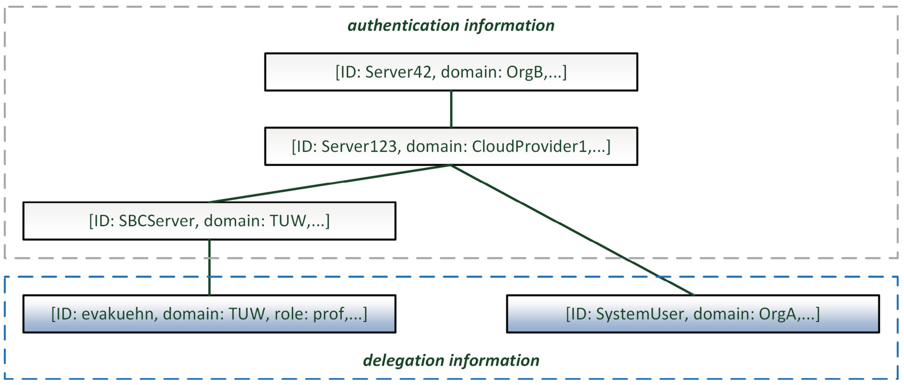
authentication information
[ID: Server42, domain: OrgB,...]
[ID: Server123, domain: CloudProvider1,...]
[ID: SBCServer, domain: TUW,...]
[ID: evakuehn, domain: TUW, role: prof,...]
[ID: SystemUser, domain: OrgA,...]
delegation information

Fig. 3. Subject tree example for delegation.

peers of different companies and has also earlier authenticated the owner of P_1 as "SystemUser". This runtime then provides its authentication data together with the forwarded claims of the other principals to the final target runtime, which is owned by the runtime user "Server42" from organization "OrgB". The target peer has to specify if it trusts such a subject tree via its access control rules. For the textual representation of a subject tree, we depict authentication chain edges with the "@" symbol. The example subject tree can thus be abbreviated as "$(SystemUser\ \mathbf{for}\ (evakuehn@SBCServer))@Server123@Server42$".

3.2 Rule-Based Authorization

Based on the proposed delegation concept, a decentralized authorization mechanism can be defined, where for every peer an access control policy is specified by its owner. Each policy consists of a set of access control rules that need to be evaluated to form an access decision. It determines which entries can be written to as well as read or removed from a container. As all interactions in the Peer Model are based on entries, this protects access to a peer's internal state and its services. Due to the data-centric modeling of service invocations, requests as well as their responses can be authorized. Rules may not only apply to regular peer containers, but also to the meta containers that specify sub-peers and wirings. Thus, a peer owner may allow trusted subjects to inject their own logic into the peer, which supports the management of multitenant environments.

Fine-grained access control policies should exceed the expressiveness of simple access control lists on peers or containers. Therefore, we adapt a policy language from our previous work on secure space containers [9] to domain-specific assumptions introduced by the Peer Model. This approach is inspired by the XACML standard [13], which provides a declarative, XML-based language for expressing access control policies based on authenticated subject attributes, used operations, accessed content, and context-dependent conditions. Rules may either permit or deny a specific access request, whereas combination algorithms are used to determine a final result. However, XACML policies are rather complex and often not very comprehensible. As during the execution of a flow usually several peers managed by different owners are involved, which need to define their own access control policies, the effect of the individual policies and their combination must be easy to comprehend.

Each access control rule in the Secure Peer Space is associated to a specific peer and consists of a unique ID, a list of affected subjects, the involved containers and operations as well as restrictions on the entry content and context information. The **subjects** field contains one or more *subject templates* that are compared with the authenticated subject of the access operation that has to be authorized. The rule applies only if the subject matches at least one of these templates. Such a template is represented by a tree where each node consists of a set of predicates on the subject properties of the corresponding principal in the subject tree. These predicates resemble the selectors used in guard and action queries and may check for equality or inequality with a specific value or use comparison operators (e.g. "$age \geq 18$"). Additionally, wildcards for single nodes ("*") and chains ("**") are supported. A subject tree node matches its corresponding node in the template if each of its predicates is fulfilled. The root node can be omitted, as it always corresponds to the local runtime user. An example template that matches the subject tree from Fig. 3 would be "([$domain = OrgA$] **for** ([$domain = TUW,\ role = prof$] **@ ****)) **@** [$domain = CloudProvider1$]". This means that the rule targets delegated access by any peer from organization "OrgA" on behalf of a professor from TU Wien, which was transmitted via a runtime managed by the organization of "CloudProvider1". The wildcard states that the creator of the rule does not care whether the professor was directly authenticated at the cloud provider or indirectly via a chain of one or more other runtimes. Thus, it is not guaranteed that this authentication data was relayed only along trusted nodes. However, it could be assumed that the cloud provider is already responsible for doing these kind of checks. If every runtime trusts its direct predecessors in a flow to only accept input from other trusted nodes, a chain of trust can be established that allows for very simple subject templates.

The **resources** field specifies the container(s) for which the rule applies, while the **operations** field distinguishes between three access types: **read**, **take**, and **write**. Read access is relevant for copy guards, while take privileges are required by move guards. Write permissions are necessary for actions, including those that inject entries into remote containers via the DEST property.

The optional **scope** field states for which kind of entries the rule is valid. This is expressed via the same query mechanism as used by guards and actions, however without the count parameter. Thus, the rule's scope may be restricted to a certain entry type or to entries with specific coordination properties. A combination of queries using disjunction, conjunction and negation is possible, thus enabling complex rules. The optional **condition** field allows restrictions based on the current state of the peer, which depicts context information that may depend on previous interactions. It consists of one or more *condition predicates* that can be combined using disjunction, conjunction and negation. Each predicate consists of a container name and a query as used by the scope mechanism. If at least one entry in the specified container matches the query, the predicate is fulfilled. The rule only applies if the combination of predicates is satisfied. A condition may, e.g., be used in rules that allow access to a container only if an internal state entry in the PIC has a specific value or if a specific sub-peer is registered in the PSC. For our example, a rule that allows users to write requests to the PIC of P_1 may be defined as follows:

SUBJECTS: $[role = prof]$ @ $[ID = SBCServer,\ domain = TUW]$
RESOURCES: PIC
OPERATIONS: *write*
SCOPE: R
CONDITION: $PIC \triangleright D\ [peerCount > 0]$

The subject template matches any professor that was authenticated via a specific runtime user from TU Wien. The scope limits permissions to entries of type R, while the condition checks that the coordination property "*peerCount*" of the directory entry D in the PIC has a value greater than zero, which prevents access when no peers are available that are able to process tasks.

In contrast to XACML, we support only "permit" rules, which simplifies the combination of rules for finding an access decision. If access to a specific entry for a given subject is not permitted by at least one active rule, it is automatically denied. Due to the fine-grained rules and the possibility to specify hierarchic security policies (one for each sub-peer), most access restrictions can still be expressed easily. A rule permits access if the subject tree of an operation matches any given subject template, the operation type and the accessed container are included in the rule, the condition (if available) evaluates to true and the written or selected entries fulfill the scope query. If no scope query is specified, access to the whole container is granted. Access control is transparent for wirings: Guards only select entries from the subset of entries the wiring owner is permitted to access, while other entries are not visible by the query mechanism. Thus, a wiring cannot distinguish if an entry does not exist or if the subject does not have the necessary permissions. When actions try to write entries to a container, the set subject must be allowed to write each of them, otherwise the action is skipped.

In order to let permissions not only depend on the context of the peer, but also on the context of the accessing subject, we introduce *context variables* that can be used instead of any coordination property value in scope and condition queries as well as subject templates. These variables are prefixed with a "$" symbol followed by a name that represents a specific subject property for the current access. To simplify the specification of such constraints, the original issuer of a flow (i.e. the leftmost leaf in the subject tree) is aliased as "originator". This allows, e.g., to specify that the domain of a runtime user in the authentication chain must be the same as the domain of the original issuer of a request ("$domain = \$originator.domain$" in the subject template), or that entries may only be accessed on behalf of the principal that was responsible for writing them ("$originator = \$originator$" in a scope query).

To reduce the number of rules and prevent users from being locked out of their own peers, peer owners may implicitly access their own containers without any restrictions. However, if delegation is used (peer owner on behalf of another subject), explicit access control rules are required, which supports restrictions based on the identity of the delegating subject.

3.3 Secure Runtime Architecture

The security concept can be integrated into the Peer Space runtime architecture, resulting in the Secure Peer Space. The runtime hosts peers and is responsible for storing entries, executing wirings, handling remote communication as well as enforcing authentication and authorization. Fig. 4 shows the architecture of this middleware framework. As the runtime has input and output stages in the form of a remote communication interface, it can be represented in the meta model by a *runtime peer* that is owned by the runtime user. Incoming entries are written to the PIC and dispatched to the corresponding peer, either according to the address in the given DEST property or using explicit wirings. Similarly, entries that need to be sent to a remote runtime are written to the POC. Top-level peers and wirings can be added by writing to the PSC and WSC of the runtime peer.

Authentication is done by an *authentication manager* that intercepts received entries before writing them to the PIC of the runtime peer (or to a meta container, e.g. when adding peers). It is responsible for verifying the credentials that were sent with each entry. If authentication is successful, the authenticated attributes are attached to the entry's subject tree, otherwise the entry is discarded. The authentication manager may be configured to accept attributes from one or more identity providers, which may be restricted to specific domains to prevent that an identity provider for one organization issues identities associated with a competitor. The security mechanism is independent of the used authentication method, whose details are therefore out of scope of this paper. Identity providers may either directly communicate with the authentication manager or indirectly via information already included in the received entry. As the level of trust in a subject may depend on the used authentication method and the responsible identity provider, information about the authentication context is also included in the subject properties by the authentication manager.

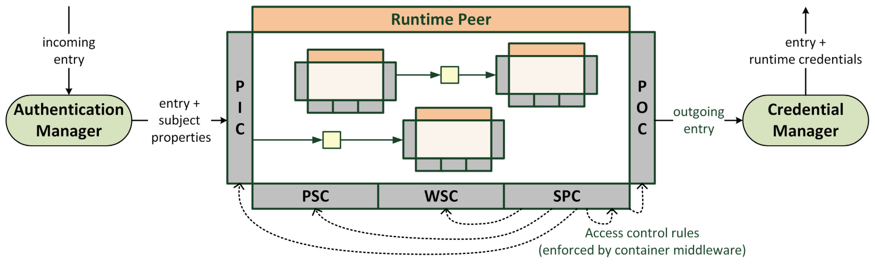
incoming entry
Authentication Manager
entry + subject properties
Runtime Peer
P I C
P O C
outgoing entry
Credential Manager
entry + runtime credentials
PSC
WSC
SPC
Access control rules (enforced by container middleware)

Fig. 4. Secure Peer Space runtime overview.

Before an entry is sent to a remote runtime, the *credential manager* attaches the credentials of the runtime user to the outgoing entry (e.g. by using a digital signature), so that the remote runtime's authentication manager may correctly authenticate the entry. We assume that the used communication channels are cryptographically secured to ensure confidentiality and integrity for entries.

The access control policy for each peer is managed via an additional meta container named *Security Policy Container* (**SPC**) that holds the access control rules as individual entries. The policy can be managed by writing and removing rules via (dynamic) wirings, which allows for flexible permission changes. Rules may target any container of the corresponding peer, including the SPC itself. Thus, a peer owner may grant administrator privileges to another subject by specifying a rule that permits (possibly restricted) access to the SPC.

The policy for the entire Peer Space is defined in a hierarchic way via the SPCs of the runtime peer and all hosted (sub-)peers. An administrator may define general rules that specify who is trusted to communicate with the Peer Space, while the owners of the hosted peers may restrict access to their services to specific subjects. If necessary, more fine-grained permissions may be set in sub-peers. As entries are always passed up or down along this hierarchy, each involved stakeholder can control what kind of interactions are allowed. This also applies when using the remote communication mechanism, as the runtime moves an entry with set DEST property recursively through the POC of each parent peer until the POC of the runtime peer is reached. On the receiving side, the entry recursively passes the PICs of child peers until the destination is reached. The entry owner must be permitted to write to each of these containers.

The enforcement of the access control policy can be embedded into the container implementation using a mechanism described in [14]. Each container access is intercepted and evaluated with regard to the active policy stored in the responsible SPC. The runtime determines rules with matching subject, operation and container. Then, conditions are evaluated by querying the specified containers. Finally, for all remaining (i.e. applicable) rules the scope is evaluated, either on the set of written entries or on the entire container (for read/take access). The container operation is only performed if it is allowed according to the specified rules, whereas denied entries are treated as invisible for query operations.

4 Secure Coordination Patterns

Coordination patterns provide reusable design solutions to recurring problems for the interaction of autonomous components. Due to the high decoupling provided by the Peer Model, complex applications may be designed by configuring and composing such "building blocks" consisting of several peers and their coordination logic [15]. In the following, we outline two coordination patterns with respective access control rules as examples for the usage of the Secure Peer Space.

4.1 Request-Response with Cloud Service

For the first example, we address a request-response scenario, where a client peer sends a request entry (*Req*) to a server peer, which generates a response entry (*Resp*) that is returned to the client. The server is hosted at a generic cloud platform that may act as runtime peer for several server peers managed by different principals. The Peer Model representation of this pattern is depicted in Fig. 5, which also shows the relevant security attributes of the peer owners and the subjects for the individual operations. For the sake of simplicity, we assume that all principals share a domain. To prevent misuse, several access control rules need to be defined by the respective peer owners. For the server peer *AppPeer*1 (owned by *App*1), the following rules may be defined in its SPC:

SUBJECTS: $[role = Client]$
RESOURCES: *PIC*
OPERATIONS: *write*
SCOPE: *Req*

SUBJECTS: $[ID = App1]$ **for ****
RESOURCES: *POC*
OPERATIONS: *write*
SCOPE: *Resp*

The first rule allows clients to invoke the server using a request entry, while the second rule indicates that there are no restrictions on whose behalf a response entry may be sent. Taking a request from its PIC is implicitly allowed for the peer owner *App*1. The same (or more general) rules must also be set by the owner of the runtime peer *CloudRTP*. To enable the dynamic adaptation of server peers via meta containers, additional rules have to be specified by the runtime user, which are not detailed here. Finally, the client, which owns runtime peer *ClientRTP*, may want to ensure that a server only sends a response entry when it acts on the client's behalf. That is achieved using the following rule:

SUBJECTS: $([role = Server]$ **for** $[ID = User1])$ @ $[ID = Cloud1]$
RESOURCES: *PIC*
OPERATIONS: *write*
SCOPE: *Resp*

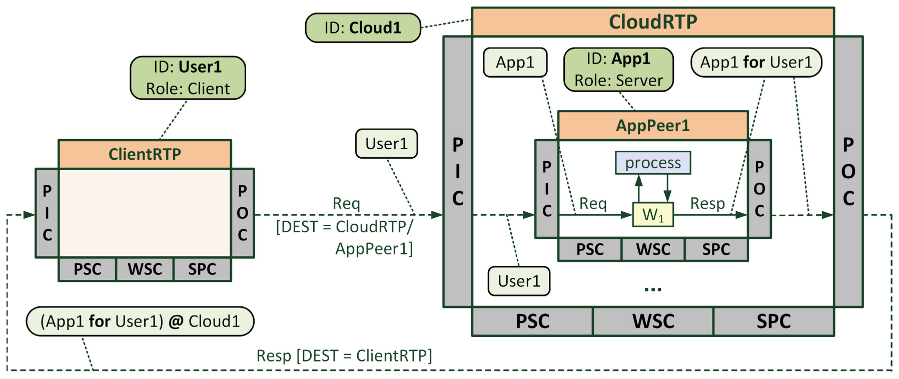

Fig. 5. Cloud-based request-response pattern with responsible subjects.

4.2 Data Exchange via Shared Memory

The second example outlines a shared memory on a server peer that can be accessed by several nodes that want to exchange data. A possible use case would be home automation, where several devices may want to exchange sensor data to collaboratively achieve a task. Fig. 6 depicts such a scenario, where two robot peers that are part of an alarm system application share their sensor data via a central home server. The following rule regulates access to this storage peer:

SUBJECTS: $[role = Node]$
RESOURCES: PIC
OPERATIONS: $write,\ read,\ take$
SCOPE: $Data\ [originator.app = \$originator.app]$

It ensures that any peer with role $Node$ may write and retrieve data entries. However, data may only be accessed within the same application, which prevents, e.g., that the entertainment system reads sensitive data from the alarm system. This is achieved via a context-aware selector in the scope, which ensures that only entries sharing the subject property *app* with the entry owner are considered. Optionally, the rule may be extended with additional conditions, e.g. to ensure that the application is currently active based on a status entry. Read and take access are realized via dynamic wirings. In the example, $RobotPeer2$ retrieves the data of $RobotPeer1$ by writing the wiring specification for W_1 to the WSC of $StoragePeer$. This requires additional rules that allow write access for nodes to the WSC and the POC, as well as a rule on $RobotPeer2$ that permits the dynamic wiring to respond via the PIC of the robot peer, using "$[ID = Robot2]$ @ $[role = Server]$" as subject template to express trust in the server.

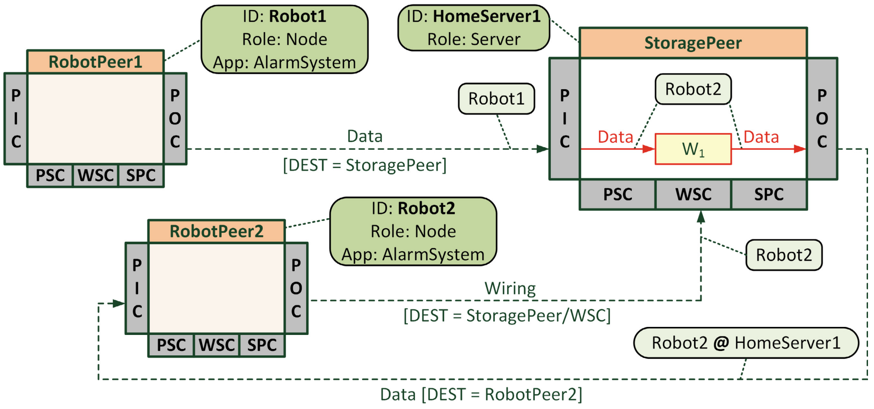

Fig. 6. Shared memory pattern in home automation scenario with responsible subjects.

5 Discussion and Related Work

The proposed Secure Peer Space architecture offers an abstraction for modeling secure collaboration of autonomous peers in distributed systems with fine-grained, context-aware permissions for stored data as well as service invocations. Due to its explicit trust model, it supports multiple identity providers and heterogeneous authentication mechanisms. Authorization is decentralized, as each user regulates access to its own peers based on the trustworthiness of the request issuer, the delegated principals, and the involved runtimes. Also for multitenant environments, security constraints can be expressed in a natural way due to the support for access control on meta-model operations and the hierarchical layers of protection provided by the nested peer structure. As policy administration is bootstrapped using meta containers, a holistic security model for collaborative scenarios is provided that allows for specifying flexible access control rules targeting all kind of peer interactions and administrator tasks.

The underlying Peer Model separates coordination and computation in a business process, while the proposed concept adds access control in a decoupled way. Consequently, each component can be administered independently allowing reuse of secured peers in different workflows. As a tradeoff, administrators have to know the basic functionality of involved peers. While simpler ways of handling authorization are possible, the expressiveness of our model enables the definition of complex constraints that would otherwise have to be included directly in the application code. If such complexity is not desirable, an actual implementation could simplify rules, e.g. by defining the scope only by means of entry types and omitting conditions. It is also possible to model a peer that dynamically changes rules based on a high-level security policy, which may be easier to comprehend than the combination of individual rules in different policy containers.

Chained delegations are already an established concept for access control in cross-organizational communication networks. Earlier approaches [3,12,16] focus on providing cryptographic assurance that delegation is authorized by delegating principals along the chain, thus preventing malicious nodes from acting on behalf of arbitrary principals. PERMIS [4] supports decentralized ABAC and RBAC, where authorization of delegation chains is based both on policies of the identity provider (included in the credentials) and of the receiving node. Trust-related access control rules on the target define which attributes specific identity providers are trusted to issue. In the Secure Peer Space, we use subject templates to combine such rules with regular privilege-based rules. As we focus on fine-grained authorization instead of authentication, the receiver of an entry is responsible for evaluating if the delegation chain appears trustworthy. However, an authentication mechanism that ensures a delegation was also authorized by the originator could be included in a similar way as in PERMIS.

Other related systems emphasize the importance of a decentralized authorization approach, but do not support delegation chains. P-Hera [5] provides secure content hosting for P2P infrastructures, where resource and data owners can dynamically establish trust via fine-grained XACML rules. Similar to the Secure Peer Space, each subject may express its own constraints based on its role in the network. Opyrchal et al. [17] suggest an access control model for a publish-subscribe middleware in pervasive environments, where owners can authorize other users to subscribe to their events. Fine-grained rules can be specified using conditions on attributes of the request, the addressed event, and context information. As in our approach, secure policy administration is bootstrapped via rules that allow users to modify a policy. LGI [1] provides a secure message exchange mechanism for open groups of distributed agents. Members may independently define their own access control policy, as long as it conforms to a common coalition policy. Expressive Prolog-based rules can be specified that may depend on the current state of the interaction. The Secure Peer Space also supports a hierarchy of policies via nested peers. A shared policy may be enforced by an administrator that manages distributed coalition peers, which contain subpeers owned by the respective members. TuCSoN [18] supports coordination via distributed tuple spaces connected in a tree topology, where gateway nodes control visibility and authorization of their children. Such an architecture may also be enforced with the Secure Peer Space by only allowing access if an entry was forwarded by a gateway peer. Like in our runtime architecture, access control is realized using features of the space-based middleware itself. Similar to our approach, SMEPP [19] is a service framework on top of a tuple space abstraction, where service requests and replies are modeled as data entries in shared spaces. However, access control is based on groups and thus rather coarse-grained.

6 Conclusion

We have presented a decentralized access control model that addresses interactions of autonomous peers which do not fully trust each other. Each principal may specify its own security policy that governs access to its data and services. By using the Peer Model as a data-driven abstraction for collaborative workflows and meta-level operations, we are able to specify a wide range of security constraints via fine-grained rules on (meta) containers that depend on properties of the authenticated subject, the accessed entries and context data. Due to an expressive delegation mechanism, trust-based rules can be specified that depend not only on the request originator, but also on the trustworthiness of users that have forwarded the request and security attributes of other principals.

For the proposed middleware architecture, we are currently developing a prototype that should be applicable for different domains, including cloud architectures and P2P networks. Even in situations where the runtime itself is not feasible, e.g. due to limited resources of embedded systems, the secured Peer Model version can still be helpful in the design phase to model all kind of security constraints in a unified way. Future work will also involve additional research on secure coordination patterns in order to provide an extensive pattern catalogue that covers the most relevant forms of interaction in collaborative scenarios.

Acknowledgments. We would like to thank Lukas Bitter, Stephan Cejka, Thomas Hamböck, and the anonymous reviewers for their helpful comments.

References

1. Ao, X., Minsky, N.H.: Flexible regulation of distributed coalitions. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 39–60. Springer, Heidelberg (2003)
2. Ahmed, T., Tripathi, A.R.: Security Policies in Distributed CSCW and Workflow Systems. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans **40**(6), 1220–1231 (2010)
3. Gomi, H., Hatakeyama, M., Hosono, S., Fujita, S.: A delegation framework for federated identity management. In: 2005 Workshop on Digital Identity Management, pp. 94–103. ACM (2005)
4. Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: A modular authorization infrastructure. Concurrency Computation Practice and Experience **20**(11), 1341–1357 (2008)
5. Crispo, B., Sivasubramanian, S., Mazzoleni, P., Bertino, E.: P-Hera: scalable fine-grained access control for P2P infrastructures. In: 11th International Conference on Parallel and Distributed Systems, vol. 1, pp. 585–591. IEEE (2005)
6. Kühn, E., Craß, S., Joskowicz, G., Marek, A., Scheller, T.: Peer-based programming model for coordination patterns. In: De Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 121–135. Springer, Heidelberg (2013)
7. Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C.: Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In: 8th International Conference on Autonomous Agents and Multiagent Systems, vol. 1, pp. 625–632. IFAAMAS (2009)
8. Carriero, N., Gelernter, D.: Linda in context. Communications of the ACM **32**(4), 444–458 (1989)
9. Craß, S., Dönz, T., Joskowicz, G., Kühn, E., Marek, A.: Securing a Space-Based Service Architecture with Coordination-Driven Access Control. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications **4**(1), 76–97 (2013)
10. Yuan, E., Tong, J.: Attributed based access control (ABAC) for web services. In: 2005 IEEE International Conference on Web Services, pp. 561–569. IEEE (2005)
11. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **29**(2), 38–47 (1996)
12. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: theory and practice. ACM Transactions on Computer Systems **10**(4), 265–310 (1992)
13. Moses, T.: eXtensible Access Control Markup Language (XACML) Version 2.0. Standard, OASIS (2005)
14. Craß, S., Dönz, T., Joskowicz, G., Kühn, E.: A coordination-driven authorization framework for space containers. In: 7th International Conference on Availability, Reliability and Security, pp. 133–142. IEEE (2012)
15. Kühn, E., Craß, S., Schermann, G.: Extending a peer-based coordination model with composable design patterns. In: 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 53–61. IEEE (2015)
16. Gasser, M., McDermott, E.: An architecture for practical delegation in a distributed system. In: IEEE Computer Society Symposium on Research in Security and Privacy, pp. 20–30. IEEE (1990)

17. Opyrchal, L., Prakash, A., Agrawal, A.: Designing a publish-subscribe substrate for privacy/security in pervasive environments. In: 2006 ACS/IEEE International Conference on Pervasive Services, pp. 313–316. IEEE (2006)
18. Cremonini, M., Omicini, A., Zambonelli, F.: Coordination and access control in open distributed agent systems: the TuCSoN approach. In: Porto, A., Roman, G.-C. (eds.) COORDINATION 2000. LNCS, vol. 1906, pp. 99–114. Springer, Heidelberg (2000)
19. Benigni, F., Brogi, A., Buchholz, J.L., Jacquet, J.M., Lange, J., Popescu, R.: Secure P2P programming on top of tuple spaces. In: 17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 54–59. IEEE (2008)