SmartSociety

Hybrid and Diversity-Aware Collective Adaptive Systems
When People Meet Machines to Build a Smarter Society

Grant Agreement No. 600584

Deliverable 7.1 Working Package 7

Technical Report – SmartCom Design

Dissemination Level (Confidentiality):[1]	*PU*
Delivery Date in Annex I:	*31/12/2014*
Actual Delivery Date	*31/12/2014*
Status[2]	*F*
Total Number of pages:	57
Keywords:	*virtualization, communication, middleware, SmartCom*

[1]PU: Public; RE: Restricted to Group; PP: Restricted to Programme; CO: Consortium Confidential as specified in the Grant Agreeement
[2]F: Final; D: Draft; RD: Revised Draft

Disclaimer

Full project title:	SmartSociety: Hybrid and Diversity-Aware Collective Adaptive Systems: When People Meet Machines to Build a Smarter Society
Project Acronym:	SmartSociety
Grant Agreement Number:	600854
Number and title of work-package:	7 Programming Models and Frameworks
Document title:	Technical Report – SmartCom Design
Work-package leader:	Hong-Linh Truong, TUW
Deliverable owner:	Ognjen Scekic
Quality Assessor:	Daniele Miorandi, UH

List of Contributors

Partner Acronym	Contributor
TUW	Philipp Zeppezauer, Ognjen Scekic, Hong-Linh Truong

Executive Summary

This document presents the SmartCom architecture, components, and application programming interfaces. Section 1 takes a look at the general architecture of the system and how the functionalities are mapped and handled within the system and its various components. Afterwards, Section 2 examines how SmartCom handles messages between the platform, the applications, the middleware, collectives, and the peers, and how messages look like. Section 2.4 presents the APIs of the components that have been described in Section 1. Finally, Section 2.5 outlines some more complex algorithms that are used by the system.

Table of Contents

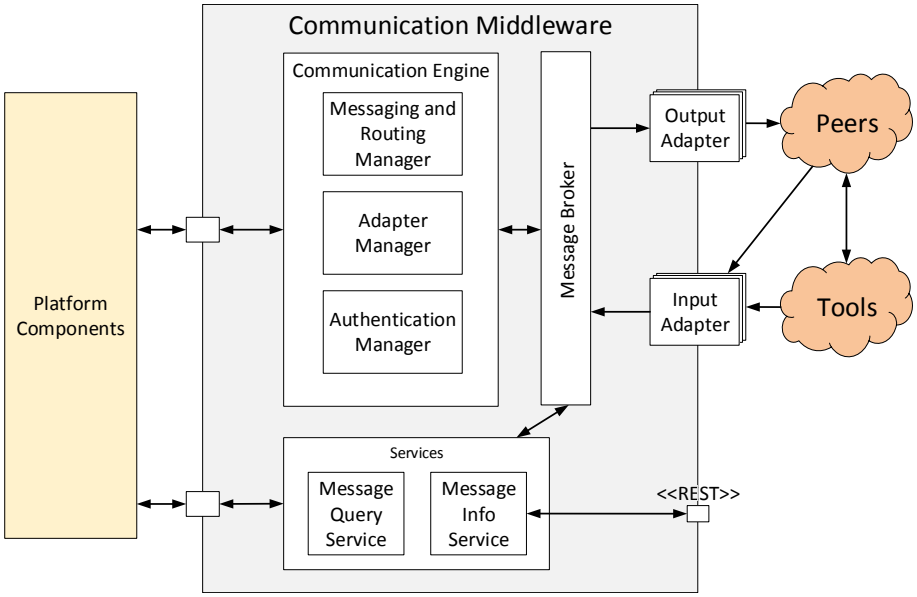

Figure 1: Overview of the communication middleware. Platform Components are part of the HDA-CAS platform and peers/tools are external to the system.

1 Architecture

Figure 1 presents a conceptual overview of SmartCom's internal architecture. In a typical use-case, the SmartSociety platform components on the left hand side initialize the communication by sending messages to peers and collectives on the right hand side via SmartCom. We denote the messages flowing from the SmartSociety platform in the direction of peers as *Output Messages*. Peers can reply to received messages by sending messages via SmartCom. Additionally they can use external, third-party tools (e.g., uploading a file to a file server) which are monitored by SmartCom (i.e., the system checks regularly if there are updates/changes). Messages originating from the peers being passed on to SmartCom for delivery are referred to as *Input Messages*.

SmartCom consists of four groups of components:

- **Adapters** are used to handle and virtualize the actual communication with peers and tools over communication channels (e.g., sending an email or making a REST call) from the rest of SmartCom and the platform. The technology that is used by adapters to send and receive messages depends on the actual implementation and is abstracted from the rest of the system by providing a common interface for all adapters. This is important to provide virtualization of peers. Furthermore, this

allows the adaptation of technological advancements in communication.

- **Communication Engine** consists of components that provide the core functionality of the system. They are responsible for the handling of messages intended to be sent to or received from peers and tools. They resolve current collective members and initialize communication. Additional functionalities, like message authentication, management and execution of adapters and routing of messages are also provided by the components of the Communication Engine.

- **Services** provide additional information on communication aspects to the peers and platform components. Furthermore, they can be used, e.g., to derive metrics (such as average response time) and profiles for peers.

- **Message Broker** is used to decouple the execution of various middleware components to achieve scalability (i.e., multiple components listen for messages on a single queue). Furthermore, the queues of the broker are used for the routing of messages which is determined by the Messaging and Routing Manager of the Communication Engine.

The following sections describe the internal structure and further details of the components mentioned above. A detailed diagram of all the components is presented in Figure 2.

1.1 Adapters

This section discusses the technical aspect of adapters within SmartCom. In general there are two different types of adapters: **Output Adapters** and **Input Adapters**. They differ in their behavior as the Output Adapter is only allowed to send output messages to peers and the Input Adapter is only allowed to receive input messages from peers. The reason behind this distinction is that the two types are handled differently. Whereas Output Adapters are shared among all applications running on the platform, Input Adapters are usually created by applications and are dedicated to receive input messages for the application that created the adapter. Their behavior is application specific compared to the behavior of Output Adapter which is considered as peer specific.

This difference in behavior is also expressed in the way they are created. Output Adapters are only registered in the system as adapter types (e.g., an email adapter that sends emails to peers) and their lifecycle (i.e., registration, creation, execution, and removal) is handled by the Adapter Manager (discussed in Section 1.2.1). On the other hand,

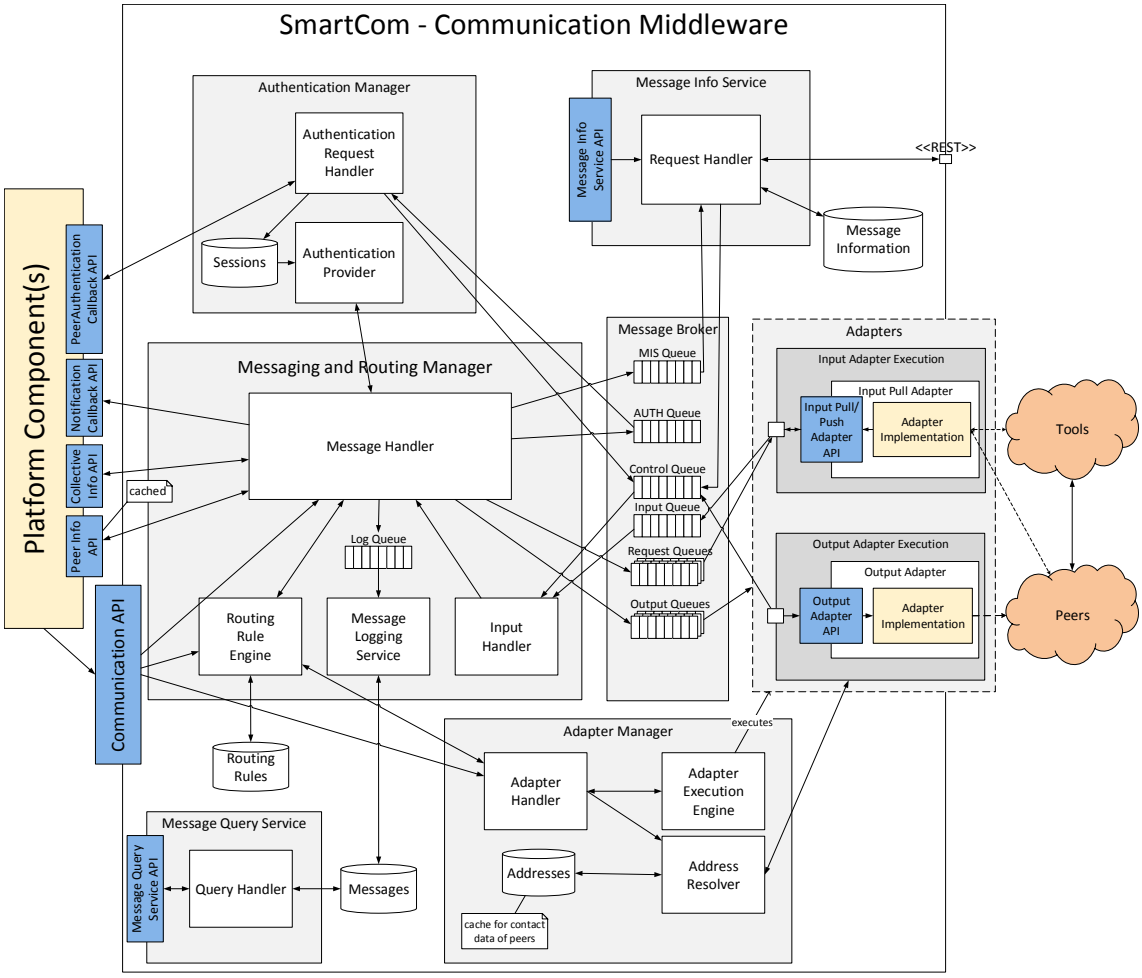

Figure 2: Detailed view of the communication middleware.

Input Adapters are created by applications running on the platform and are passed to SmartCom because they might require special configuration. Consider an Input Adapter that monitors a folder of a FTP server, such an adapter would require a path to be specified as well as some additional information like username and password. This data has to be provided by applications because this information is application specific. Therefore, Input Adapters are not shared among different applications.

1.1.1 Output Adapters

Output Adapters are responsible for sending messages from SmartCom to peers. There are two categories of Output Adapters: *Stateful Output Adapters* and *Stateless Output*

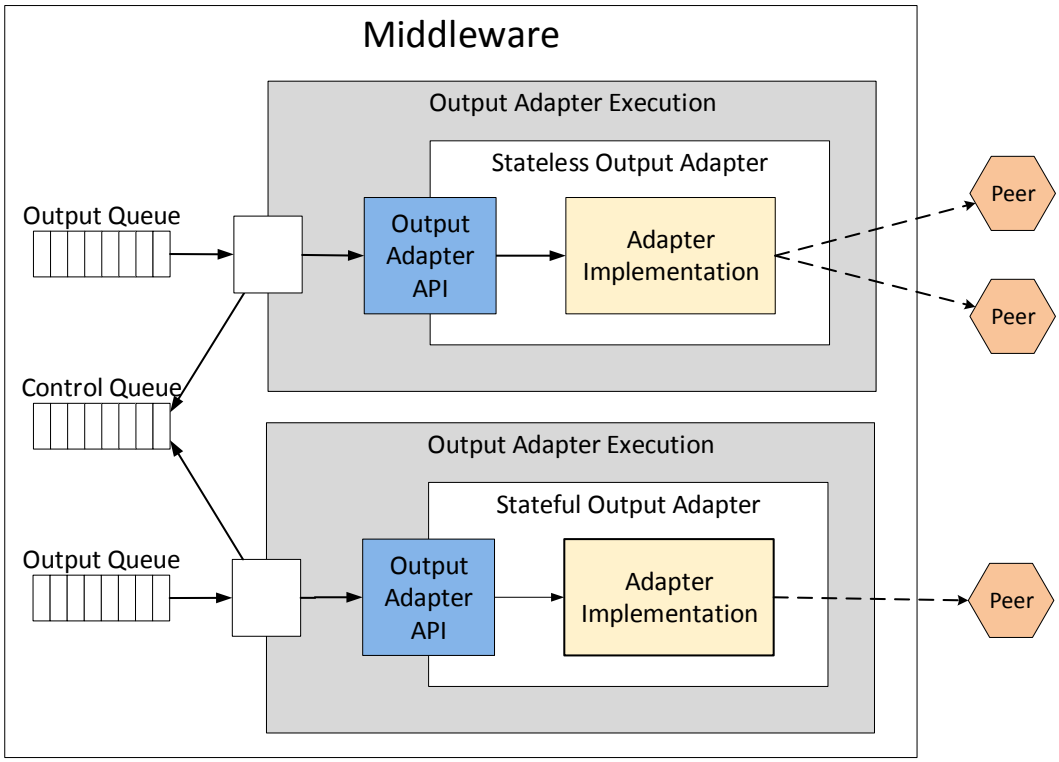

Figure 3: Concept of the Output Adapter

Adapters. Stateful Output Adapters instances are created per peer, whereas single Stateless Output Adapters instances are used to send messages to different peers. Both categories have to be provided with peer specific contact data, such as an email address or an URL. The necessary data to contact a peer with an Output Adapter is provided by the Adapter Manager at each invocation. Additionally, Stateful Output Adapters are provided with this data also at the beginning of their lifecycle, because they might require to maintain additional conversational data based on this peer information. Due to the different lifecycles (further details below) they are provided with that data during the creation of the adapter. Figure 3 presents the internal structure of both categories of Output Adapters. The boxes labeled 'Adapter Implementation' indicate the actual implementation of the corresponding adapter (e.g., code that issues a REST call). It can also be observed that both adapters have their own Output Queue but they share one Control Queue. Output Queues are adapter specific queues which are used for outgoing messages that still have to be handled. The Control Queue is used to notify SmartCom that the

sending was successful or of an error. Instances of both categories have to implement the `Output Adapter API` (see Section 2.4.2).

The lifecycle of an Output Adapter depends on whether it is stateful or stateless. Both Stateful Output Adapters and Stateless Output Adapters, have to be registered in the system by calling the Adapter Manager (see Section 1.2.1).

- In case of a Stateless Output Adapter, the adapter is instantiated and executed immediately because they are shared among peers. This approach has the advantage that there is no need to instantiate the adapter of a specific type at any point in the future, which eliminates the need for synchronization and locking to ensure that there is only a single instance of this adapter. Note that further scaled out instances and the primary instance of the same adapter are considered as a single instance from the conceptual point of view.

  The disadvantage of this approach is that resources, such as computation time and memory, are assigned to the adapter even if the adapter is not used to communicate with peers. Nevertheless, due to the usually limited resource consumption of Stateless Output Adapters, this disadvantage is acceptable.

- In case of a Stateful Output Adapter the adapter is only instantiated on-demand because there is an instance per peer that uses this adapter for communication. If there are many peers using an adapter type (e.g., an email adapter) there are also many adapter instances which causes a higher resource consumption compared to Stateless Output Adapters. Hence, creating them on-demand and – if they have not been used for some time – removing them reduces this resource consumption.

Adapters of both categories are usually just removed if the appropriate method of the Adapter Manager is called. However, instances of Stateful Output Adapters could be discarded earlier to save resources in case they have not been used for some time. Removing an Output Adapter means that the corresponding communication channel cannot be used by SmartCom to interact with peers. For example, removing an Output Adapter that sends emails results in not being able to contact any peer using emails unless another Output Adapter handling emails is registered.

1.1.2 Input Adapters

Input Adapters are responsible for either waiting for input or for actively checking for input from peers or tools. Input Adapters can be implemented using a push or pull mechanisms. Adapters using push are called *Input Push Adapters*. They are notified by the external tool/communication channel of new developments (e.g., a new mail in the mailing list) via a push notification. On the other hand, *Input Pull Adapters* are handled and executed by the Adapter Execution Engine. The pull is triggered in a certain interval or based on a programmed request (e.g., a peer has only one hour to send a file to a FTP server. After the time runs out, the pull adapter checks if there is a file available). This request is expressed by putting a corresponding message in the Request Queue of a pull adapter. This message instructs the adapter to execute a pull. Figure 4 presents the internal structure of both Input Adapter categories. Instances of both categories push the received message to the Input Queue. Input Push Adapters have to implement the `Input Push Adapter API` (see Section 2.4.2), instances of Input Pull Adapters have to implement the `Input Pull Adapter API` (see also Section 2.4.2).

The lifecycle of Input Adapters is managed entirely by SmartSociety platform applications/components. Input Adapters are created and removed by applications because their configuration is application specific.

The lifecycle of Input Push Adapters is special, because after adding them to the Adapter Manager they have to register a technology-specific handler that is responsible for the reception of push notifications. This handler has to be destroyed again when the adapter is removed. For example, an adapter using a server socket has to register it at the beginning and destroy it at the end of its lifecycle.

1.2 Communication Engine

The Communication Engine is the core of the execution system of SmartCom and is responsible for the communication between the platform, the applications, and the peers using messages and adapters. The Communication Engine consists of the Adapter Manager, Authentication Manager, and the Messaging and Routing Manager. The interactions of the subcomponents can be examined in Figure 2.

Messages that should be sent to peers are passed to the Messaging and Routing Manager which decides based on internal routing rules how to forward messages (i.e., which component/adapter handles the message). Messages are sent to and received from the

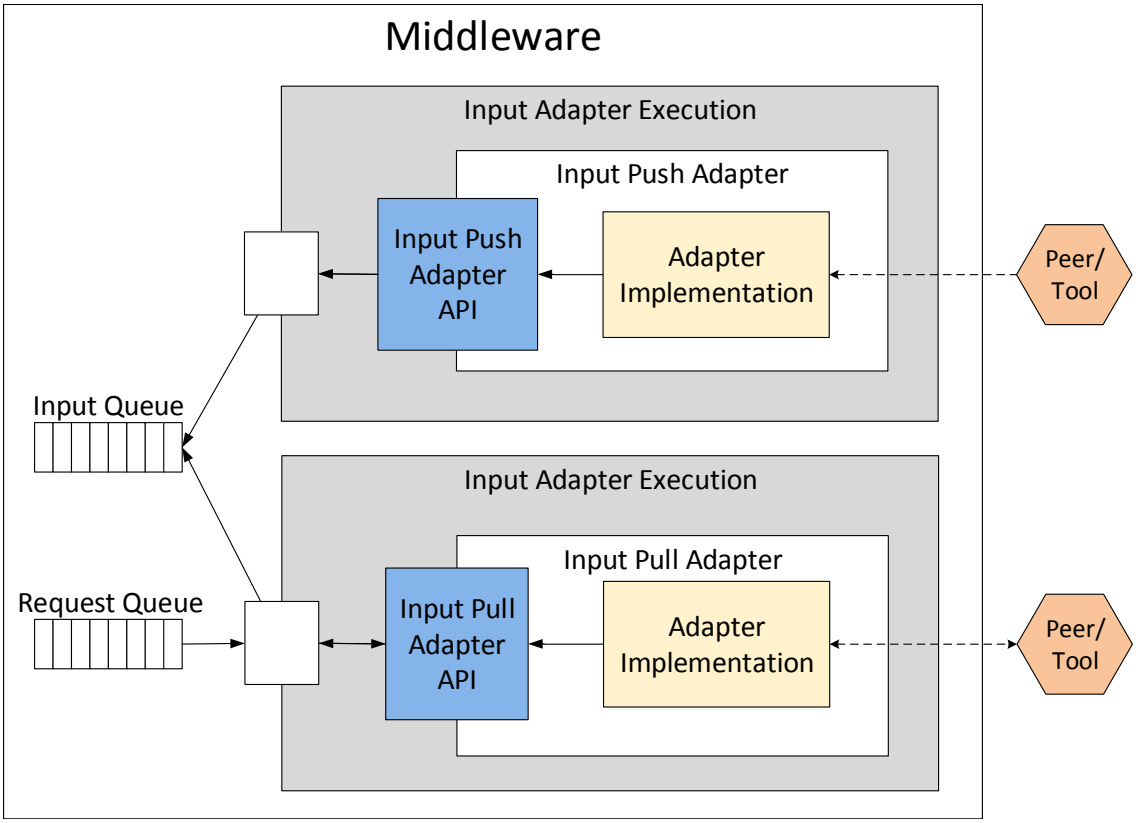

Figure 4: Concept of Input Push Adapters and Input Pull Adapters.

peers using corresponding Input and Output adapters(described in Section 1.1), which are created, managed and executed by the Adapter Manager. The Authentication Manager is responsible to verify the authenticity of a peer and to provide a security token to peers that allows SmartCom to verify the sender of a message. These components are described in the following sections.

1.2.1 Adapter Manager

The Adapter Manager is responsible for the lifecycle management (i.e., registration, creation, initialization, execution, and removal) of adapters. It consists of the following subcomponents: Adapter Execution Engine, Adapter Handler, Address Resolver and multiple Adapter Executions. Figure 5 shows the internal structure of the Adapter Manager and how the subcomponents interact with each other.

The **Adapter Handler** manages the lifecycle of Output Adapters. Both categories of

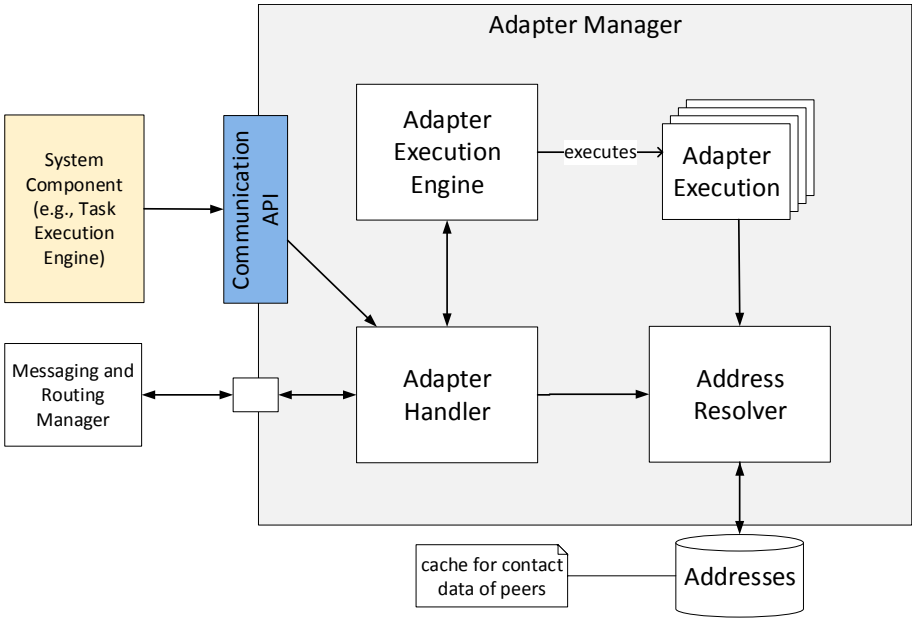

Figure 5: Subcomponents of the Adapter Manager and interaction with other internal and external components.

Output Adapters are registered with the Adapter Handler. Stateless Output Adapters are instantiated immediately, while Stateful Output Adapters just remain registered. In case a message has to be sent to a peer using a stateful adapter, the Adapter Handler creates an instance of the adapter for the recipient of the message and passes its reference to the Messaging and Routing Manager. The reference of the adapter represents the address of the adapter-specific Output Queue of the Message Broker the adapter pulls messages from. The Adapter Handler prevents the instantiation of multiple stateful adapter for a single peer. The selection of the required adapters for a communication with a peer is based on the *Peer Channel Addresses*[3]. The Peer Channel Address is the internal representation of a communication channel used by a peer. Input Adapters are not registered at the Adapter Handler. Platform components and applications have to create instances of Input Adapters themselves and pass these instances to SmartCom using the Communication API (see Section 2.4). Input Adapters either receive messages from peers or tools directly via push notification or they check an external tool (e.g., a folder on a FTP server or a mailing list) regularly if there is a new message represented by a new resource (e.g., a new file) available.

To support scaling out to handle big workloads, each Output Adapter initially listens

[3]They consist of a unique name (e.g., Email) and a list of parameters (e.g., an email address)

for new messages on a single, adapter specific Output Queue of the Message Broker (see Section 1.3). Scaled out instances of an adapter pull messages from the same queue which allows multiple instances to handle messages of the queue concurrently. Conceptually, the initial instance of the adapter and the scaled out instances are considered a single adapter instance, because there is no difference in semantics, just an increase in performance. Therefore, we will not differentiate between these instances in this description. Note that Stateful Output Adapters and Stateless Output Adapters differ in their behavior regarding scalability. Since each Stateful Output Adapters instance is associated with a peer there is hardly any need for scaling out these instances because a higher workload only occurs in rare cases. Since they are shared, Stateless Output Adapters have to be scaled out much more often, especially if there are lots of peers using the communication channel handled by the adapter.

Instances of both types of adapters are executed by the **Adapter Execution Engine**. Therefore, every adapter is assigned to an **Adapter Execution** which handles its execution. The Adapter Execution retrieves messages from queues, determines the address information of communication channels of peers, calls the appropriate methods from the `Adapter APIs` (see Section 2.4) and publishes messages to queues. The behavior of the Adapter Execution depends on whether it handles an Input Adapter or an Output Adapter. Executions of Output Adapters retrieve messages from the Output Queue and initiate the communication. Executions for Input Pull Adapters wait for pull requests in the Request Queue and initiate a pull request upon reception of a message. Input Push Adapters handle the adapter's execution on their own, they are not assigned to Adapter Executions. The details of algorithms for adapter lifecycle management can be examined in Section 2.5.1.

Peer Channel Addresses for instantiated adapters are stored in the **Addresses Data Storage**. These addresses are needed by adapters to be able to contact a peer. The **Address Resolver** is responsible to resolve address requests by Adapter Executions. When an adapter is sending a message to a peer, the Adapter Execution provides the address of that peer by querying the Address Resolver. The data storage acts as a cache for Peer Channel Addresses to speed up the execution of adapters because the Peer Channel Addresses are usually managed by platform components and calling them regularly might be a limiting factor to performance and throughput.

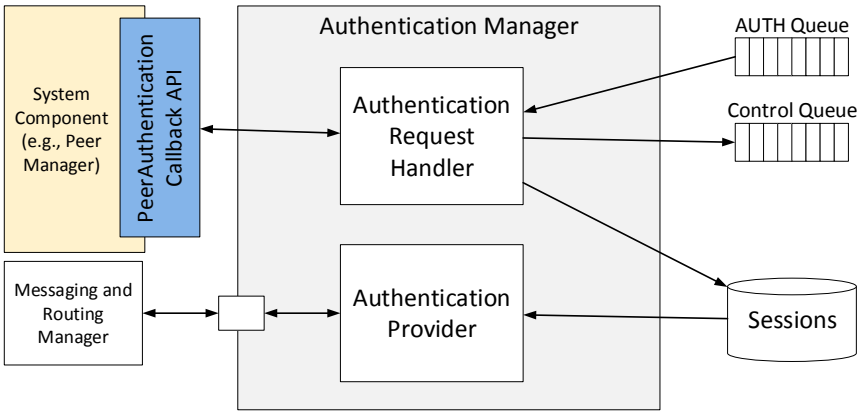

Figure 6: Concept of the Authentication Manager

1.2.2 Authentication Manager

The Authentication Manager is used to authenticate peers and verify the authenticity of their messages in the system. Authentication request messages (see Section 2.3) are dropped in the AUTH Queue by the Messaging and Routing Manager and are collected by the **Authentication Request Handler**. This handler interacts with the `Peer Authentication Callback API` (see Section 2.4.3) to get information on the peer and to authenticate the peer (using the credentials provided in the message). After the successful authentication, the manager creates a security token that can be used by peers and SmartCom to provide security features (e.g., message authentication or message encryption). This token is only valid a certain period of time. The time period between the creation of the token and the invalidating thereof is called *session*. The result of the authentication is passed to the Control Queue in form of a response message. The **Authentication Provider** can be used by the Messaging and Routing Manager to verify the authenticity of a message – if required. Figure 6 presents the internal structure of the Authentication Manager.

The Authentication Manager uses a **Session Data Storage** to handle the sessions of peers. Sessions consist of a *session token* that can be used by peers to authenticate messages, and a timestamp. If a message arrives with a token of an invalid session, the peer has to be informed to renew its token. Such messages should be discarded or at least retained until the peer authenticates itself again.

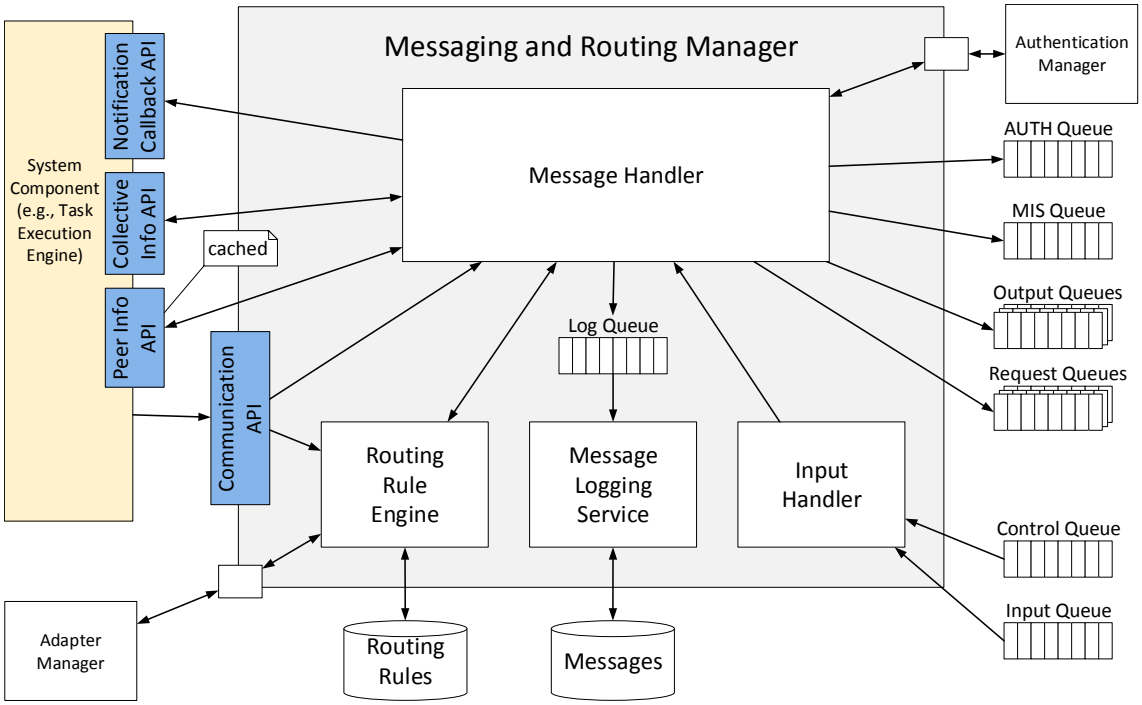

Figure 7: Concept of the Messaging and Routing Manager

1.2.3 Messaging and Routing Manager

The Messaging and Routing Manager is responsible for handling of internal and external messages of the system. Figure 7 presents the internal structure and the communication with external components of the Messaging and Routing Manager. Messages are sent to peers, collectives, or components by this component.

Upon reception of a message the **Message Handler** handles the messages according to the type of the receiver of the message. If the receiver of the message is a peer, the Message Handler determines the corresponding adapter(s) that should be used for the communication. If there are no adapters available, new ones have to be created. Therefore, the Message Handler queries the `Peer Info API` (see Section 2.4.3) to retrieve the peer profile information which contains the communication channels used by the peer. Since this information does not change often, it can be cached to improve the performance of further requests. Subsequently, the Adapter Manager is instructed to create new adapters according to the peer's preferred communication channels and delivery policies. After the successful creation of adapters, the message is put in the corresponding Output Queues

of the adapters.

If the receiver of the message is a collective, the current members of the collective have to be determined in order to initiate the communication. Therefore, the Message Handler queries the `Collective Info API` (see Section 2.4.3). After receiving the members of the collective, a new message is created and sent to every member using the procedure described in the previous paragraph.

If the receiver of a message is an internal or external component, the Message Handler directly forwards it to the corresponding component either by putting it into a special queue or by notifying the `Notification Callback API` (see Section 2.4.3) of the corresponding component. The Messaging and Routing Manager uses routing (described later in this section) to be able to determine the corresponding component. If no receiver is set for a message, the Message Handler notifies a platform component using the `Notification Callback API`, because there is no possibility to determine a receiver due to the stateless nature of SmartCom.

Platform components can register themselves with the Messaging and Routing Manager to receive notifications upon messages through the `Notification Callback API`. All registered callbacks implementing this API are invoked whenever a message arrives that cannot be handled by SmartCom.

Besides the primary recipient of a message, further recipients of messages can be determined based on *Routing Rules*, which are handled by the **Routing Rule Engine**. Rules can be added by platform components and applications to implement special communication patterns, or to simplify the communication and reduce overhead. For example, in an application a message of a specific subtype is always transferred to a software service; the message can be forwarded directly to the software instead of sending it the application first.

The **Input Handler** pulls incoming input messages (e.g., a response from a peer) from the Input Queue and incoming control messages (e.g., a communication error message) from the Control Queue. Both messages are forwarded to the Message Handler that determines their destination. It is possible to scale out the Input Handler to improve the performance of the handling of input and control information.

The **Message Logging Service** is responsible for persisting of all sent and received messages to a database. This information can be used to debug SmartCom, or to analyze the data for determining incentives or constructing provenance graphs. Stored messages

can be retrieved by using the Message Query Service (see Section 1.4.1).

Routing There are two types of routing available in the proposed SmartCom. The first type of routing is purely internal and represents the determination of corresponding adapter(s) that have to be used for the communication with a peer. No routing rules are involved in this process. This activity involves that the Adapter Manager be instructed to instantiate new adapters if there are none available for a specific peer. This type of routing also has to keep track of changes in the peer information of a peer because this might result in the recreation/removal of previously instantiated adapters. Additionally, the delivery policy (described later on) of a peer has to be tested for changes because this might also trigger instantiations/removals of adapters using the Adapter Manager.

The second type of routing determines further recipients of a message based on the following properties of messages: type and subtype, receiver, and sender. This routing information can be added by providing *Routing Rules* which are stored in the Routing Rule Engine. The resulting routing defines additional recipients of the message which can either be peers, collectives, or internal and external components. Note that routing rules with an empty recipient are not allowed due to obvious reasons. Routes are determined by matching the properties of the message to properties of the routing rules, whereas setting properties of the routing rules `null` matches every corresponding property of the message. The route is determined by the properties in ascending order. First, the type is determined, afterwards the subtype, then the receiver and finally the sender of the message. Because null matches anything, it is not allowed to provide a routing rule with the type, subtype, receiver, and sender being all null. This restriction prohibits that all messages of the system are forwarded to a single peer.

The second type of routing adds flexibility to the system in terms of communication. It allows applications to implement special communication patterns – e.g., a monitoring peer that logs all the messages sent to a specific collective without being part of the collective.

1.2.4 Handling of Policies

SmartCom handles two types of policies. The first type are peer-specific privacy policies which have to be considered when sending a message to a peer. Privacy policies might restrict the sending of messages based on their properties or at a certain time (e.g., during the night) which means that the sending has to be aborted. Privacy policies of peers

are managed outside of SmartCom and are retrieved by calling the `Peer Info API` (see Section 2.4.3).

The second type of policies are the delivery policies. There are multiple levels where they have to be considered and enforced. They are described in the following:

1. The first level of delivery policy enforcement is on the message level. At this level it is possible to specify how messages are delivered, initially there is just an option to determine whether a successful sending of a message is acknowledged or not. In general the delivery policies on this level and any other level are not restricted to this behavior. They can be easily extended in further versions of SmartCom.

2. The second level of delivery policies is concerned with the peer delivery policies. Besides specifying the preferred communication channels in peer's profile (in an external component), these policies can also specify how a peer is to be contacted using these addresses. The options TO_ALL (all addresses are used), TO_ANY (any address is used) and PREFERRED (preference is expressed by the order of addresses) are currently provided. The enforcement of these policies is handled by the Messaging and Routing Manager. When a message is sent to a peer, the manager registers a handler that listens for acknowledgement messages from the corresponding adapters which indicate a successful sending. Unsuccessful sending is indicated by a communication error message of the adapter. The TO_ALL policy requires that all adapters are able to successfully send the message, whereas TO_ANY requires at least one adapter to be successful. The delivery policy PREFERRED fails if the message could not be sent to the preferred adapter of the peer. In case of an unsuccessful delivery based on the chosen delivery policy, a failure message is forwarded to the sender of the initial message.

3. The third level of delivery policies is concerned with the sending of messages to collectives. This information about delivery policies is provided by the `Collective Info API` (see Section 2.4.3) and defines how messages should be sent to members of the collective. Options include TO_ALL_MEMBERS and TO_ANY. Upon sending a message to a collective, there is also a handler registered to enforce the policy. The behavior on this level is similar to the one on the peer level but successful sending is indicated by a successful policy enforcement on the peer level. TO_ALL_MEMBERS means that the sending of messages has to be successful for each peer based on the peers' delivery policies, if one of these policies fails, the enforcement on the collective

level fails too. On the other hand the TO_ANY policy only requires the sending to one peer to be successful.

Note that the failure of a policy enforcement is always reported to the sender of the initial message, the success case is just reported if required by the policy on the message level of the initial message.

1.3 Message Broker

The purpose of the Message Broker is to decouple the executions of the various components of SmartCom. Furthermore, it is used to implement the first type of routing (see Section 1.2.3) to send messages to the correct adapters that have to forward it to the peers. Some of the queues of the Message Broker have already been mentioned in previous chapters, in the following we briefly describe all available queues that are used within the system:

- **Control Queue**: contains messages that have been sent by internal components and are needed to control the internal flow of messages, to forward results of internal service invocations (e.g., answer of an authentication request), to indicate (communication) errors, or to enforce delivery policies.
- **Input Queue**: a single queue that is filled with input messages of peers by all Input Adapters. These messages are handled by the Messaging and Routing Manager according to the specified receivers and additional routing rules.
- **Output Queues**: contain the output messages that should be handled by adapters to send a message to a peer over a communication channel. There is exactly one output queue for each Output Adapter.
- **Request Queues**: used to force Input Pull Adapters to perform a pull. There is one queue for each of these adapters so that they can be notified separately to pull for new input.
- **AUTH Queue**: a special queue for messages that are intended for the Authentication Manager. Messages in this queue are authentication request messages (see Section 2.3) which consist of the username and password so that peers can be authenticated.

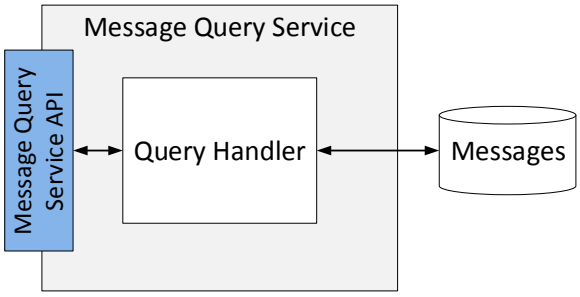

Figure 8: Concept of the Message Query Service

- **MIS Queue**: a special queue for messages that are intended for the Message Info Service. These messages are usually request messages (see Section 2.3) for information about a certain message indicated by a certain type and subtype.
- **Log Queue**: intended for all messages that are handled within SmartCom and that have to be logged. Messages in this queue are consumed by the Message Logging Service of the Messaging and Routing Manager which saves the messages to a database.

1.4 Services

1.4.1 Message Query Service

The Message Query Service provides an interface to query sent and received messages. Figure 8 presents the internal structure of the Message Query Service. The **Query Handler** is responsible for the handling of queries and the execution of queries in the database. This service can be used to query all internal and external messages that have been handled by SmartCom.

1.4.2 Message Info Service

The Message Info Service provides information about messages based on their type and subtype. It is used by peers to get information on how to interpret a message and how to respond. Furthermore, it provides a human-readable description of the message's structure and contents, as well as its semantic meaning and relation with other messages. This service could also be improved to return an explanation how to interpret the message in a machine-readable way. The prototype provides a simple textual description that the worker can fetch to interpret the message semantics, especially with respect to related

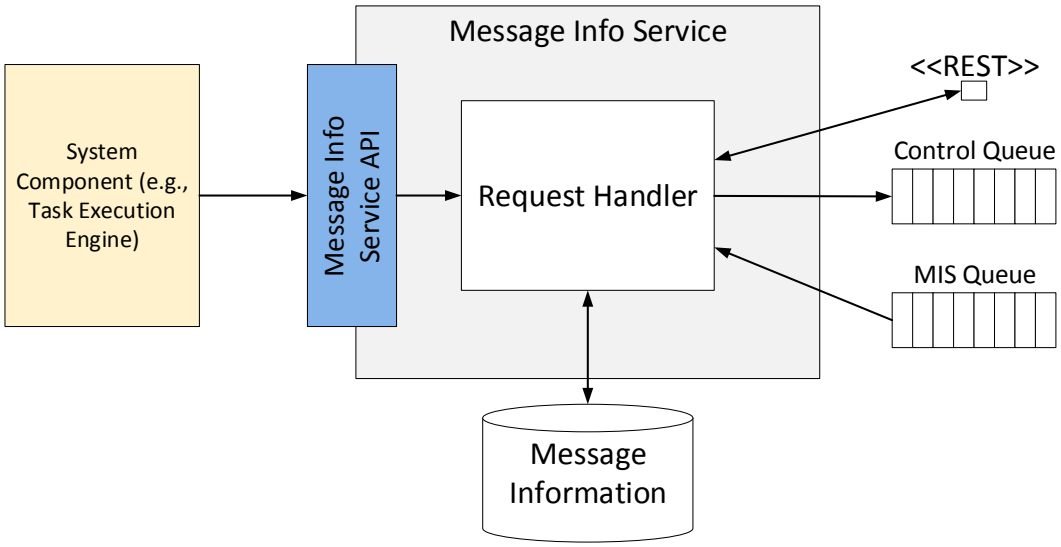

Figure 9: Concept of the Message Info Service

messages. The service maintains a database to store the message information which is updated through platform components. Figure 9 shows the internal structure of the Message Info Service and how it is connected to the queues.

The service can be used by a peer either by sending a message info request (see Section 2.3) to an adapter or by invoking a REST service that provides the corresponding data. Since this data is application specific, it has to be provided by the application using the `Communication API` (see Section 2.4.2).

2 Messages

SmartCom exchanges messages with platform components, the adapters as well as with some internal components (i.e., the Authentication Manager and the Message Info Service). The following section takes a look at how these messages look like, how the routing of messages is handled and some predefined message types are discussed. Note that further message types and subtypes can be defined by programmers of applications for a HDA-CAS. The semantics of such message types and subtypes depend on the application that created them.

2.1 Message Structure

The following section presents the structure of messages that are used within SmartCom. The structure is quite similar to the FIPA ACL Message Structure [1], but some properties have been removed and others added to fit the requirements of SmartCom.

Each message consists of several mandatory and optional fields. The most important fields of a message are the Id of the message, the sender, the type and subtype. These and further fields are discussed and described in Table 1. Listing 1 outlines a simple message containing instructions for a task in the JSON format.

```
{
  "id": "2837",
  "type": "TASK",
  "subtype": "REQUEST",
  "sender": "peer291",
  "receiver": "peer2734",
  "conversation-id": "18475",
  "content": "Check the status of system 32"
}
```

Listing 1: Example message with instructions for a task

After receiving a message, Output Adapters are responsible to transform them to the appropriate technology-related and peer-understandable representation and send the message using a communication channel. On the other hand, Input Adapters are responsible for the transformation of received messages of a technology-related message format (e.g., email) to an internal message. Messages that are related to a specific execution of an application are required to have a execution-dependent conversation-Id, otherwise it is not possible to associate a message with the corresponding execution. Note that SmartCom does not use the conversation-Id internally, this functionality has to be provided by platform component.

2.2 Routing of Messages

The routing of messages is handled by the Messaging and Routing Manager according to rules based on the message's type, subtype, receiver and sender. The order (type, subtype, receiver, sender) also defines the priority, which means that the type has the highest priority and the sender the lowest. Further information on routing can be found in Section 1.2.3.

Field	Description
Type	This field defines the high-level purpose of the message (e.g., control message, input message, metrics message, etc.). This field is especially important for the routing of messages within the system.
Subtype	This field is defined by the component that is in charge of the message (i.e., it is component specific). The subtype combined with the type of the message defines the purpose of the message. The subtype can also be used by programmers of applications to define custom message types for their application.
Message-Id	A global unique identifier is assigned to every message within the system by the Messaging and Routing Manager.
Sender-Id	The sender-Id specifies the sender of the message (can be a component, peer, etc.). Sender-Ids are unique within the systems. Sender-Ids are either predefined in case of an internal component or are assigned by platform component.
Receiver-Id (o)	The receiver-Id specifies the receiver of the message (can be a component, peer, collective). Can also be empty if the receiver is not clear.
Conversation-Identifier (o)	Denotes the system identifier for the conversation. This identifier can be used by platform components to map the message to the actual execution instance of an application. For example: application A is executed twice at the same time: A_1 and A_2. The conversation-Id is used to associate the messages with the right executions A_1 or A_2. If there is no conversation (e.g., for internal messages), the conversation-Id can also be empty.
Content (o)	Defines the content of the message including instructions and data that are needed to execute the message. This can be empty in case of simple messages (e.g., acknowledge messages).
TTL (o)	Time to live. Defines a time interval in which a message is valid. For example: a peer has one hour to post pictures in a folder of a FTP server, after this time SmartCom stops looking for pictures in the folder and creates an error message if there are no pictures.
Language (o)	Denotes the language of the message. This can be a natural language, like English or German, as well as a computer format like binary. The initial intention of this field are logging and debugging purposes. In future versions a translation service could be introduced that makes use of this field.
Security-Token (o)	The security token can be used to guarantee the authenticity of messages or to encrypt the content of the message.
Delivery-Policy (o)	Specifies the delivery policy of the message. This field can be used to specify if the sender wants an acknowledgement in case of a successful sending of the message.
RefersTo (o)	This field can be used to specify that this message refers to another message.

Table 1: Structure of messages. Optional fields are marked with (o).

2.3 Predefined Messages

These messages are needed for special purposes, like authentication, or to indicate specific behavior (i.e., an acknowledged message) or exceptional cases and errors. The following sections describe these predefined messages and define their intended usage in the system. The subtypes of the messages are defined in the corresponding rows within brackets and in capital letters.

2.3.1 Control Messages

Control messages are exchanged within SmartCom and are exposed to the application. Control messages are always indicated by the message type *CONTROL*. Their intention is to indicate specific control behavior (e.g., acknowledgement of a message) or exceptions during the communication. They are described in detail below. Table 2 presents the various subtypes.

Message	**Description**
Acknowledge (*ACK*)	This message is sent by the output adapter if the message has been successfully sent to the peer. Note that this does not imply peer's acceptance of the contents of the message, but is used to implement functionalities such as read receipts. This message is not sent if the programmer requires a fire-and-forget sending behavior (i.e., she doesnt care if it actually has been delivered).
Error (*ERROR*)	An error message that indicates a generic error. This message is handled based on the routing rules.
Communication Error (*COMERROR*)	This error message indicates an error during the communication. This is reported to the sender of the initial message.
Timeout (*TIMEOUT*)	This message indicates that a time out has appeared in the system and that the message couldn't be delivered in time or there was no response within a certain time.

Table 2: Predefined subtypes of Control Messages.

2.3.2 Message Info Messages

These messages are handled by the Message Info Service (see Section 1.4.2) and are intended for requests of message information by peers over dedicated input adapters and for the reply of such a request. All such messages are required to have the message type *MESSAGEINFO*. Table 3 presents the two subtypes of message info messages.

Message	Description
Message Info Request (*REQUEST*)	Request by a peer to the Message Info Service for information on how to interpret and handle a given message based on its type, and subtype.
Message Info Response (*REPLY*)	Response of the Message Info Service to a peer that contains information on how to interpret and handle a given message.

Table 3: Message Info Request and Reply Messages.

2.3.3 Authentication Messages

Authentication messages are used to perform authentication of a peer in the system and provide him with a security token that is valid for a specific time period (internally called session). Such messages are handled by the Authentication Manager (see Section 1.2.2) which interacts with platform component to verify the identify of a peer. Further information can be found in section 1.2.2. Authentication messages always have the type *AUTH*. AuthenticationRequest messages are sent by peers to the system whereas the three other messages (AuthenticationResponse, AuthenticationFailed, AuthenticationError) are sent back from SmartCom to the peer. Table 4 describes the used subtypes.

Message	Description
Authentication Request (*REQUEST*)	Authentication request message of a peer that contains its credentials. The Authentication Manager queries platform component to verify the peer's credentials. After the successful verification, a security token is created and sent to the peer.
Authentication Response (*REPLY*)	Response message for an authenticate request message from SmartCom to the peer. It contains a security token that can be used in further requests to verify the identity of a peer.
Authentication Failed (*FAILED*)	Special response for an authenticate request message from SmartCom to the peer that indicates that the authentication failed. The purpose of this message is to distinguish between the cases of a failed authentication and a authentication error on the basis of the message's subtype.
Authentication Error (*ERROR*)	Special response message for an authenticate message from SmartCom to the peer that indicates that there was an error during the authentication of the peer. Such an error might be that, for example, no external platform component is available that can verify the credentials.

Table 4: Authentication Messages.

2.4 Application Programming Interfaces (APIs)

The following section takes a look at the API of SmartCom. First, we examine the public entities that are needed to interact with the system. Afterwards, we take a look at the callback entities that are needed by SmartCom to get required information for the communication. Finally, the interfaces and their methods are described in detail to get an understanding on how to interact with the system.

2.4.1 Data Structures

Table 5 presents the data structures that are exchanged between SmartCom and the platform components. They are mainly used by the public entities described in Section 2.4.2, and the callback entities described in Section 2.4.3.

2.4.2 Public Entities

Table 6 describes the interfaces that are exposed by SmartCom to clients. These entities are required to interact with the system and receive response. The interfaces and their methods are described in the following sections in detail.

Communication API This section discusses the main API for the interaction with peers, collectives, and SmartCom for the purpose of communication. It provides methods to start the interaction with collectives and peers, and also defines methods to extend and manipulate the behavior of SmartCom. Figure 10 presents the Communication API in UML notation.

public Identifier **send**(Message message) throws CommunicationException Send a message to a collective or a single peer. The method assigns an Id to the message and handles the sending asynchronously, i.e., it returns immediately and does not wait for the sending to succeed or fail. Errors and exceptions thereafter are sent to the Notification Callback API (see Section 2.4.3). Optionally, received acknowledgments are communicated back through the Notification Callback API. The receiver of the message is defined by the message, it can be a peer, a collective, or a component. If the receiver is not set, the message will be sent back to the

Entity	Description
Identifier	Defines an identifier object that distinguishes between different types (peer, collective, component, message) and Id combinations.
Message	Message that is exchanged between applications, SmartCom and peers. There are also internal messages that are just handled between SmartCom components, or applications and SmartCom components. See Section 2 for details.
RoutingRule	Defines a rule of how messages should be handled within SmartCom. This feature can be used to improve the handling of messages and increase the performance. A common use case is that a response message from peer A of a specific type is always be sent to peer B. See Section 2.2 for details.
PeerChannelAddress	Defines an address for a communication channel of a peer that can be handled by a specific adapter. It contains a list of parameters that can be used by an adapter to contact the peer (e.g., an email address). The number of parameters, their syntax and semantic meaning depend on the adapter. See Section 1.1 for details.
QueryCriteria	An entity that can be used to specify the criteria of a query. It is created using the Message Query Service. After specifying the criteria, a call can be made to query the database.
PeerInfo	Provides communication related information about a specific peer such as the used communication channels (PeerChannelAddresses), delivery policies defined by the peer as well as privacy policies that restrict the communication behavior. A peer is identified by an Identifier object.
CollectiveInfo	Provides the members of a specific collective as well as the collective's delivery policy. A collective is identified by an Identifier object.

Table 5: Domain model and data structures of the SmartCom.

Entity	Description
Communication	Main entity that is used for the communication with SmartCom. New messages are sent using this interface and it also allows to register new adapters and routing rules.
OutputAdapter	Adapter that is responsible to send messages to peers. There are two types of OutputAdapters: stateless and stateful adapters.
InputPushAdapter	Adapters that receive messages from peers via push communication.
InputPullAdapter	Adapters that receive messages from peers via pull communication, i.e. they query the corresponding endpoint in regular intervals.
MessageInfoService	Provides information on a specific message, i.e. how to interpret the message and the relationship to other messages.
MessageQueryService	Service that allows to query persisted messages.

Table 6: Public entities of SmartCom that are used to interact with the system.

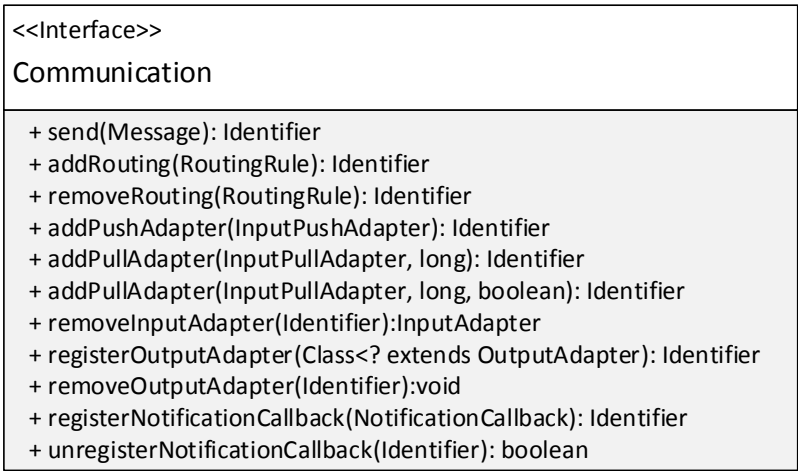

Figure 10: Communication API

Notification Callback API immediately.

Parameters

message - Specifies the message that should be handled by SmartCom. The receiver of the message is defined by the message.

Returns

Returns the internal Id of SmartCom to track the message within the system.

Throws

CommunicationException - A generic exception that is thrown if something went wrong in the initial handling of the message.

public Identifier **addRouting**(RoutingRule rule) throws InvalidRuleException

Add a route to the routing rules (e.g., route input from peer A always to peer B). Returns the Id of the routing rule (can be used to delete it). SmartCom checks if the rule is valid and throw an exception otherwise.

Parameters

rule - Specifies the routing rule that should be added to the routing rules of SmartCom.

Returns

Returns SmartCom internal Id of the rule

Throws

InvalidRuleException - If the routing rule is not valid (e.g., all fields are null).

public RoutingRule **removeRouting**(Identifier routeId)

Remove a previously defined routing rule identified by an Id. As soon as the method returns the routing rule is not applied any more. If there is no such rule with the given Id, null is returned.

Parameters

routeId - The Id of the routing rule that should be removed.

Returns

The removed routing rule or null if there is no such rule in the system.

public Identifier **addPushAdapter**(InputPushAdapter adapter)

Adds an input push adapter that waits for push notifications. Returns the Id of the adapter.

Parameters

adapter - Specifies the input push adapter.

Returns

Returns SmartCom internal Id of the adapter.

public Identifier **addPullAdapter**(InputPullAdapter adapter, long interval)

Adds an input pull adapter that pulls for updates in a certain time interval. Returns the Id of the adapter. The pull requests are issued in the specified interval until the adapter is explicitly removed from the system.

Parameters

adapter - Specifies the input push adapter.
interval - Interval in milliseconds that specifies when to issue pull requests.
Can not be zero or negative.

Returns

Returns SmartCom internal Id of the adapter.

public Identifier **addPullAdapter**(InputPullAdapter adapter, long interval, boolean deleteIfSuccessful)

Adds an input pull adapter that pulls for updates in a certain time interval. Returns the Id of the adapter. The pull requests are issued in the specified interval. If deleteIfSuccessful is set to true, the adapter is removed in case of a successful execution (i.e., a message has been received), it continues in case of a unsuccessful execution.

Parameters

adapter - Specifies the input pull adapter.
interval - Interval in milliseconds that specifies when to issue pull requests.
Can not be zero or negative.
deleteIfSuccessful - delete this adapter after a successful execution

Returns

Returns SmartCom internal Id of the adapter.

public InputAdapter **removeInputAdapter**(Identifier adapterId)

Removes a input adapter from the execution. As soon as this method returns, the adapter with the given Id is not executed any more. It returns the requested input adapter or null if there is no adapter with such an Id in the system.

Parameters

adapterId - The Id of the adapter that should be removed.

Returns

Returns the input adapter that has been removed or nothing if there is no such adapter.

public Identifier **registerOutputAdapter**(Class<? extends OutputAdapter> adapter) throws CommunicationException

Registers a new type of output adapter that can be used by SmartCom to get in contact with a peer. The output adapters are instantiated by SmartCom on demand. Note that these adapters are required to have an `@Adapter` annotation which describes the name and the type of the adapter (stateful or stateless). Otherwise an exception is thrown. In case of a stateless adapter, it is possible that the adapter is instantiated immediately. If any error occurs during the instantiation, an exception is thrown.

Parameters

adapter - The output adapter that can be used to contact peers.

Returns

Returns SmartCom internal Id of the registered adapter.

Throws

CommunicationException - If the adapter could not be handled, the specific reason is embedded in the exception.

public void **removeOutputAdapter**(Identifier adapterId)

Removes a type of output adapters. Adapters that are currently in use are re-

moved as soon as possible (i.e., current executions of communication will not be aborted and waiting messages in the adapter queue are still transmitted).

Parameters

adapter - Specifies the adapter that should be removed.

public Identifier **registerNotificationCallback**(NotificationCallback callback)

Register a notification callback that is called if there are new input messages available.

Parameters

callback - Callback for notification.

Returns

Returns SmartCom internal Id of the registered notification callback (can be used to remove it).

public boolean **unregisterNotificationCallback**(Identifier callback)

Unregister a previously registered notification callback.

Parameters

callback - Callback for notification.

Returns

Returns true if the callback could be removed, false otherwise.

Output Adapter API The Output Adapter API is used to implement an adapter that can send (push) messages to a peer. Therefore, the *push* method has to be imple-

mented. Output Adapters receive a message from SmartCom, transform this message to the adapter specific format (e.g., email) and push it to the peer over an external communication channel (e.g., send the message to a web platform or a mobile application). As described in Section 1.1 there are Stateless Output Adapters and Stateful Output Adapters. Stateless adapters are required to have a default constructor (no parameters) whereas stateful adapters can have a default constructor or a constructor with a single parameter of type PeerChannelAddress. Stateful Output Adapters are created on demand by SmartCom. Figure 11 presents the Output Adapter API in UML notation.

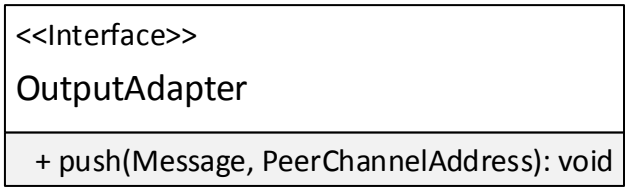

Figure 11: Output Adapter API

public void **push**(Message message, PeerChannelAddress address) throws AdapterException

Push a message to the peer. This method defines the handling of the actual communication between the platform and the peer.

Parameters

message - Message that should be sent to the peer
address - The address of the peer and adapter specific contact parameters.

Throws

AdapterException - If an exception occurred during the sending of a message

Input Push Adapter API The Input Push Adapter API is used to implement an adapter for a communication channel that uses push to get notified of new messages. The concrete implementation has to extend the InputPushAdapter class, which provides methods that support the implementation of the adapter. The external tool/peer pushes the message to the adapter, which transforms the message into the internal format and calls

the *publishMessage* of the InputPushAdapter class. This method delegates the message to the corresponding queue and subsequently to the correct component of the system that handles input messages. The adapter has to start a handler for the push notification (e.g., a handler that uses long polling) in its *init* method and remove this handler in the *cleanUp* method (e.g., a server socket).

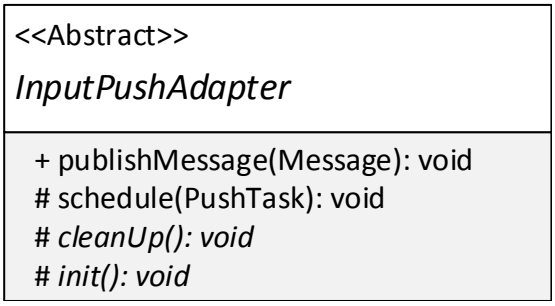

Figure 12: Input Push Adapter API

public void **init**()

Method that can be used to initialize the adapter and other handlers like a push notification handler (if needed). For example, to create a server socket that listens for connections on a specific port.

public void **cleanUp**()

Clean up resources that have been used by the adapter. Scheduled tasks using the *schedule(PushTask)* method have already been marked for cancellation, when this method is called.

protected void **publishMessage**(Message message)

Publish a message that has been received. This method has to be called when implementing a push service to notify SmartCom that there was a new message.

Parameters

message - Message that has been received.

protected void **schedule**(PushTask task)

Schedule a push task that is executed in the context of the adapter. This method should be used to reduce the resource consumption of push adapters by using an executor service. Using this method also guarantees the clean removal of adapters from the execution.

Parameters

task - Task that should be scheduled

Input Pull Adapter API The Input Pull Adapter is dedicated to pull messages from external tools or peers. For example, it can query a FTP server if there is a new file available. Instances of input adapters are always related to a single application and therefore in the context of the application, because their semantics depend on the application. Each Input Pull Adapter is executed by a single Adapter Execution of the Adapter Manager (see Section 1.2.1), which is responsible to call the *pull* method in certain intervals. Input Pull Adapters are created by applications and therefore provided with the initialization parameters by the application itself, implying a stateful adapter.

Having a stateful pull adapter has some advantages:

- The state of the communication (e.g., the corresponding execution Id of input messages) is always saved in the adapter and there is no need to save it in the Adapter Manager.
- race conditions due to the parallel execution of a single adapter are not possible because each adapter is only executed by a single thread. Therefore, no synchronization has to be applied to the adapter.

- The pull method does not require any parameters. Specific settings for adapters (e.g., an URL) can be set during the instantiation of the adapter and there is no need for a dirty parameter passing to a stateless adapter (e.g., a map or list of objects/strings).

This approach also has some disadvantages:

- Input Pull Adapters have to be created by a platform component or on higher levels (e.g., at the programming level).
- There might be a problem if too many adapters are running at the same time due to the amount of resources (i.e., memory) or required execution time. Due to the design of the Adapter Manager the Adapter Execution Engine could run on multiple machines which would eliminate or at least reduce this problem.
- Adapters have to be cleaned up properly by the creator of the adapter

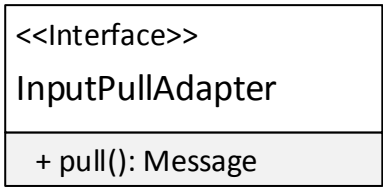

Figure 13: Input Pull Adapter API

public Message **pull**() throws AdapterException

Pull data from a predefined location. If there is no data available, null is returned.

Returns

Returns a new message or null if there is no new information.

Throws

AdapterException - If an exception occurred during the pull operation.

Message Info Service API The Message Info Service provides information about the semantics of messages, how to interpret them in a human-readable way and which messages are related to a message. Therefore, it provides methods to query message information and to add additional information to messages.

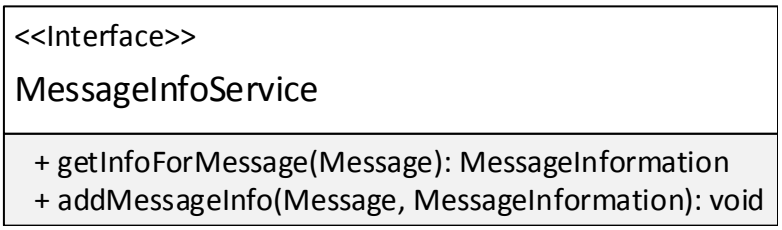

Figure 14: Message Info Service API

public MessageInformation **getInfoForMessage**(Message message) throws UnknownMessageException

Returns information about a given message to the caller. This contains how the message has to be interpreted, how it is related to other messages and which messages are expected in response to this message.

Parameters

message - Instance of a message. Must contain at least either the message Id or the message type, other parameters are optional, are used as a template.

Returns

Returns the information about a given message

Throws

UnknownMessageException - If no message of that type found or the Id of the message is not valid.

public void **addMessageInfo**(Message message, MessageInformation info)

Add information on a given message. If there already exists information for a message, it is replaced by this one.

Parameters

message - Specifies the message.
info - Information for messages of the type of parameter message.

Message Query Service API This service can be used to query the logged messages that have been handled by the system. All internal and external messages are logged by the Messaging and Routing Manager. To query the service, a QueryCriteria object has to be used that specifies the query and executes the query.

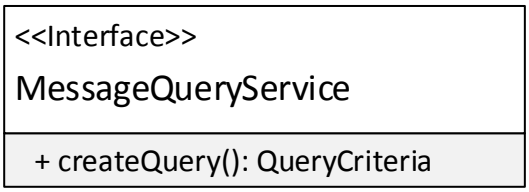

Figure 15: Message Query Service API

public QueryCriteria **createQuery**()

Creates a query object that can be used to specify the criteria for the query.

Returns

Returns a query criteria object that can be used to specify parameters and execute the query.

2.4.3 Callback Entities

Callback entities are used by the system to interact with platform components which are not part of SmartCom but that SmartCom communicates with and depends on for specific features. The corresponding components have to implement the callbacks in order to be able to communicate with them. Table 7 presents an overview of the available callback entities. These entities are described in detail in the following sections.

Entity	Description
PeerAuthenticationCallback	The Peer Authentication Callback is used by the system to verify the identity of a peer (used for authentication) and to provide security functionalities.
PeerInfoCallback	The Peer Info Callback is used to resolve peer information about a peer. This information does not change very often but is queried quite frequently, therefore retrieved data should be cached as long as the callback does not provide the required performance throughput.
CollectiveInfoCallback	The Collective Info Callback is used by SmartCom to resolve the peers that are in a collective. This information cannot be stored in SmartCom because it changes frequently, two consecutive calls might not result in the same response.
NotificationCallback	This Notification Callback is used by SmartCom to notify a platform component about messages that are not intended to be handled by SmartCom. This includes messages like task results or task-related information like communication errors.

Table 7: Callback entities of SmartCom.

Peer Authentication Callback API This callback is used to authenticate a peer within SmartCom because such information is not stored within the system but is provided by some platform component that implements this interface. After a successful authentication a session should be created to avoid calling this callback too often due to the unforeseeable performance impact.

public boolean **authenticate**(Identifier peerId, String password) throws PeerAuthenticationException;

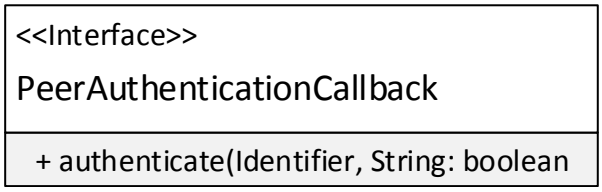

Figure 16: Peer Authentication Callback API

Authenticates a peer, i.e. checks if the provided credentials match the peer's credentials in the system.

Parameters

peerId - Id of the peer.
password - Password of the peer

Returns

Returns true if the credentials are valid, false otherwise

Throws

PeerAuthenticationException - If an error occurs during the authentication.

Peer Info Callback API This callback is used to resolve information about a peer, the so called PeerInfo. This information does not change very often but is queried quite frequently, therefore, retrieved data should be cached as long as the callback does not provide the required performance throughput.

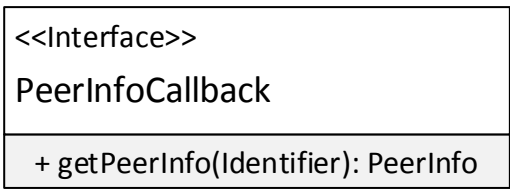

Figure 17: Peer Info Callback API

public PeerInfo **getPeerInfo**(Identifier id) throws NoSuchPeerException

Resolves the information about a given peer (e.g., provides the address and the adapter that should be used).

Parameters

id - Id of the requested peer

Returns

Returns information about a peer, such as the communication channel addresses and the preferred delivery policy.

Throws

NoSuchPeerException - If there exists no such peer.

Collective Info Callback API This API is used to provide information regarding the composition and the state of the collectives to SmartCom, in order for SmartCom to allow to platform components the functionality of addressing their messages on the collective level.

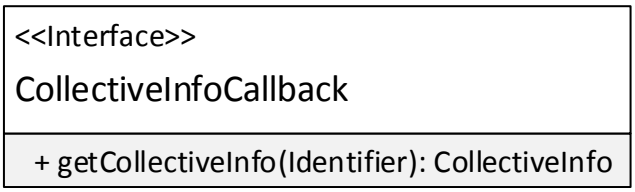

Figure 18: Collective Info Callback API

public CollectiveInfo **getCollectiveInfo**(Identifier collective) throws NoSuchCollectiveException

Resolves and returns the members of a given collective Id.

Parameters

collective - The Id of the collective.

Returns

Returns a list of peer Ids that are part of the collective and other collective related information.

Throws

NoSuchCollectiveException - If there exists no such collective.

Notification Callback API The Notification Callback is used to inform the different platform components of the messages that arrived for them (e.g., to inform the components about task results or other task-related information like an error) or that the receiver of a message could not be determined.

Since SmartCom does not save any conversational state, it is not possible to determine the right recipient if multiple platform components are implementing the Notification Callback API. Therefore, these components are required to be capable of handling (filtering) unexpected messages.

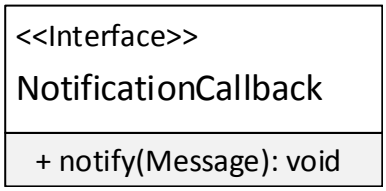

Figure 19: Notification Callback API

public void **notify**(Message message)

Notifies the corresponding callback about task results or task-relation information like an error.

Parameters

message - The received message.

2.5 Algorithms

The following section presents some important algorithms of SmartCom in pseudocode which are needed for the creation and handling of adapters, as well as the handling and routing of messages in SmartCom.

2.5.1 Creation of Output Adapters

Algorithm 2.1 describes how Output Adapters are created/instantiated in the Adapter Manager (see Section 1.2.1) based on a peer's delivery policy and the provided addresses of communication channels.

The algorithm prefers Stateless Output Adapters over Stateful Output Adapters because they have a smaller impact on the performance of the system. All or at least multiple peers share one Stateless Output Adapter, therefore, they have a lower resource usage. Contrary to Stateful Output Adapters, Stateless Output Adapters are instantiated immediately after their registration in the system, therefore, they are not instantiated using this algorithm. On the other hand, Stateful Output Adapters are instantiated per peer and on demand, because they have a higher resource usage in the system compared to Stateless Output Adapters.

Depending on the chosen delivery policy the algorithm either instantiates a single adapter (in case of the delivery policy PREFERRED) or multiple adapters (in case of delivery policies TO_ALL_CHANNELS and AT_LEAST_ONE). Note that the ordering of addresses defines the preference of communication channels (and therefore adapters) of a peer. Also note that TO_ALL_CHANNELS means all available channels, therefore, it is not an error if there is no adapter registered in the system that can handle a specific address. The intentional meaning is that the sending to all available communication channels has to succeed. The same applies for adapters that could not be instantiated due to an error. Finally the algorithm returns the internal Ids of adapters, which are required by the Messaging and Routing Manager and the Message Broker to send messages to peers. If no adapters have been found, the returned list is empty.

```
1  Function createAdapterInstances is
       input  : Peer information (peerInfo)
       output: Identifiers of the created adapters
2      addresses = peerInfo.addresses;
3      policy = peerInfo.deliveryPolicy;
4      for all address in addresses do
5          if there is a stateless adapter instance available for this address then
6              add the address of the stateless instance to the result list;
7              if policy == PREFERRED then
8                  return result list;
9              else
10                 continue with next address;
11             end
12         else if there is a stateful adapter implementation for this address then
13             if there is already an instance of that adapter for this peer then
14                 add the address to the result list;
15                 if policy == PREFERRED then
16                     return result list;
17                 else
18                     continue with next address;
19                 end
20             end
21             instantiate new stateful adapter with a unique ID;
22             add new instance to the instances of stateful adapters;
23             add new instance to the result list;
24             if policy == PREFERRED then
25                 return result list;
26             else
27                 continue with next address;
28             end
29         else
30             log that there was an unknown adapter;
31     end
32 end
```

Algorithm 2.1: Creation of adapters instances for a peer based on peer's delivery policy.

2.5.2 Handling of Messages

Messages are handled by the Messaging and Routing Manager (see Section 1.2.3). Every incoming message, regardless of whether it is from an internal component, an application or a peer is handled by the *handleMessage* function. Algorithm 2.2 depicts the function. Line 7 of the algorithm indicates the application of a routing rule which has been described in Section 2.2.

First, the message is assigned with a unique message Id which is used to track the message within SmartCom. Additional to the receiver of the message (can also be empty), further receivers - if there are any - are determined by the Routing Rule Engine based on routing rules (see Section 1.2.3). Note that the delivery policy handler is only created if the receiver of the message has been set and it is only created for the first receiver. This prevents the case of receiving an acknowledge and a communication error message for messages that have been sent to multiple receivers. Further details on the enforcement of delivery policies can be found in the following section. Finally the presented algorithm forwards the message to the corresponding receivers.

Algorithm 2.3 describes how the messages are forwarded to a collective and Algorithm 2.4 describes how the messages are forwarded to single peers. The function calls *registerCollectiveMessageDeliveryAttempt* and *registerPeerMessageDeliveryAttempt* indicate the registration of a policy handler that observes whether a delivery policy has been enforced for an outgoing message or if there was an error during communication (see the corresponding paragraph in Section 1.2.4).

First, the algorithm retrieves the collective info from the CollectiveInfoCallback directly. This object contains information about the delivery policy of the collective as well as the peers that are currently part of the collective. If required (indicated by the variable *createHandlers*) a collective message delivery attempt is registered. Thereafter the message is delivered to every peer which is currently part of the collective. Note that this membership is subject to constant change.

Similar to sending messages to a collective, the peer info are retrieved first. It consists of the delivery policy, privacy policies and contact addresses for adapters (which are not used in this algorithm). First, the algorithm checks if a message is allowed to be sent to

```
1  Function handleMessage is
       input : Message (msg)
2      if message Id is empty then
3          create unique message ID;
4      end
5      createPolicyHandlers = false;
6      if message receiver is not null then
7          add the message receiver to the receiver list;
8          createPolicyHandlers = true;
9      end
       /* check if there are further receivers                    */
10     get further receivers from the routing engine (based on routing rules);
11     add them to the receiver list;
12     if receiver list is empty then
13         send error message to NotificationCallback;
14         return;
15     end
16     for each receiver in receiver list do
17         if receiver is component then
18             forward message to component;
19             continue;
20         else if receiver is collective then
21             deliverToCollective(msg, receiver, createPolicyHandlers);  // Alg.  2.3
22         else
23             try
24                 deliverToPeer(msg, receiver, createPolicyHandlers);    // Alg.  2.4
25             catch
26                 send error message to the sender of the message;
27         createPolicyHandlers = false;
28     end
29 end
```

Algorithm 2.2: Handling of messages in the Messaging and Routing Manager.

```
1  Function deliverToCollective is
       input : Receiver (collective)
               Message (msg)
               Boolean value whether to create delivery policy handler
       (createHandlers)
2      retrieve collective info (collInfo) from CollectiveInfoCallback;
3      if createHandlers then
           // to trace the enforcement of delivery policies
4          registerCollectiveMessageDeliveryAttempt(msg, collInfo.deliveryPolicy);
5      end
6      for each peer in collInfo.peers do
7          try
8              deliverToPeer(msg, receiver, createHandlers);        // see Alg.  2.4
9          catch
10             enforceCollectiveDeliveryPolicy(new error message);
11             if collInfo.deliveryPolicy is TO_ALL_MEMBERS then
                   /* delivery failed because the massage could not be sent
                      to everyone                                          */
12                 break;
13             end
14     end
15 end
```

Algorithm 2.3: Sending messages to a collective.

```
1  Function deliverToPeer is
       input  : Receiver (peer)
                Message (msg)
                Boolean value whether to create delivery policy handler
       (createHandlers)
2      retrieve peer info (peerInfo) from PeerInfoCallback;
3      for each policy in peerInfo.privacyPolicies do
           // check if policy allows sending messages
4          if !policy.condition(msg) then
5              throw an exception;
6          end
7      end
8      if createHandlers then
           // to trace the enforcement of delivery policies
9          registerPeerMessageDeliveryAttempt(msg, peerInfo.deliveryPolicy);
10     end
11     determine list of adapters (adapterList) from routing engine;
12     if adapterList is empty then
13         throw exception;
14     end
15     for each adapter in adapterList do
16         send output message to adapter using the message broker;
17     end
18 end
```

Algorithm 2.4: Sending messages to peers.

a peer at the moment based on its privacy policies. If required (indicated by the variable *createHandlers*) a peer message delivery attempt is registered. The list consists of Ids of adapters which can send the message to this peer. Finally, using the Message Broker the message is sent to the peer over each adapter (indicated by its Id) that has been returned previously by the routing engine.

Enforcing delivery policies As described in Section 1.2.4 there are multiple delivery policies on three different levels (collective, peer and message level) which have to be enforced. Handlers for these policies are registered during the sending of messages to peers and collectives (see Algorithm 2.3 and 2.4) but the enforcement of policies is handled upon reception of acknowledge and communication error messages which are sent by adapters.

Table 8 describes how the data structure to enforce collective delivery policies might look like. The MessageID and the SenderID represent the composed key that identifies an entry. There is a policy handler for every entry that keeps track of the policy enforcement for a specific message and sender, and decides whether a policy has been enforced, if there are still results missing or if it failed. The acronym CollPolEDS is used instead of "collective delivery policy enforcement data structure" in the following.

MessageID	**SenderID**	**Policy**	**Policy Handler**
msg1	sender1	TO_ALL_MEMBERS	policyHandlerInstance1
msg2	sender2	TO_ALL_MEMBERS	policyHandlerInstance2
msg3	sender1	TO_ANY	policyHandlerInstance3

Table 8: Data structure to enforce collective delivery policies (CollPolEDS). Underlined entries indicate the composed key for each entry.

Table 9 describes the proposed data structure to enforce peer delivery policies. It looks almost the same as the CollPolEDS except that the ID of the receiver is added to the composed key. Multiple entries of this data structure might correspond to a single entry in the CollPolEDS. In case of a message being sent only to a peer, there is no corresponding entry in the CollPolEDS.

MessageID	**SenderID**	**ReceiverID**	**Policy**	**Policy Handler**
msg1	sender1	receiver2	TO_ALL_CHANNELS	policyHandlerInstance1
msg2	sender2	receiver3	AT_LEAST_ONE	policyHandlerInstance2
msg3	sender1	receiver1	PREFERRED	policyHandlerInstance3

Table 9: Data structure to enforce peer delivery policies. Underlined entries indicate the composed key for each entry.

If a message is sent to a collective, a corresponding entry is created in the CollPolEDS. For every peer in the collective an additional entry is created in the peer delivery policy enforcement data structure. Ingoing acknowledge and communication error messages from adapters are handled on the peer level first and only if that level indicates a successful or erroneous enforcement of the delivery policy, the collective level is enforced. This behavior can be observed in Algorithm 2.5 which handles the enforcement on the peer level. If there is a corresponding entry in the CollPolEDS, the enforcement is redirected to the collective level because the peer delivery policy has been successfully enforced (in case of Line 9) or there was an error during enforcement (in case of Line 19).

Algorithm 2.6 describes the delivery policy enforcement on the collective level.

```
1  Function enforcePeerDeliveryPolicy is
       input : Acknowledge or communication error Message (msg)
2      try
3          if checkPeerDeliveryPolicy(msg) then      // might throw an exception
4              entry = discardPeerPolicyEntry(msg);
5              if entry == null then        // Policy has already been enforced
6                  return;
7              end
8              if collectiveDeliveryPolicyHasEntry(msg) then
9                  enforceCollectiveDeliveryPolicy(msg);            // see Alg. 2.6
10             else if entry.messagePolicy == ACKNOWLEDGE then
11                 send acknowledgement to entry.sender;
12         end
13     catch
           /* msg can only be a communication error message                  */
14         entry = discardPeerPolicyEntry(msg);
15         if entry == null then           // Policy has already been enforced
16             return;
17         end
18         if collectiveDeliveryPolicyHasEntry(msg) then
19             enforceCollectiveDeliveryPolicy(msg);               // see Alg. 2.6
20         else
21             send communication error message to entry.sender;
22 end

23 Function checkPeerDeliveryPolicy is
       input : Acknowledge or communication error Message (msg)
24     policy = getPeerDeliveryPolicy(msg.id, msg.sender, msg.receiver);
25     if policy == null then
26         return false;                        // entry has already been evicted
27     end
28     if msg.subtyp == ACKNOWLEDGE then
29         return policy.check();
30     else
31         throw exception;          // indicates that this is an error message
32 end
```

Algorithm 2.5: Enforcing a delivery policy on the peer level.

```
1  Function enforceCollectiveDeliveryPolicy is
       input  : Acknowledge or communication error Message (msg)
2      try
3          if checkCollectiveDeliveryPolicy(msg) then // might throw an exception
4              entry = deleteCollectivePolicyEntry(msg);
5              if entry.policy == ACKNOWLEDGE then
6                  send acknowledgement to the entry.sender;
7              end
8          end
9      catch
10         entry = deleteCollectivePolicyEntries();
11         send error message to the entry.sender;
12 end

13 Function deleteCollectivePolicyEntry is
       input  : Message (msg)
       output: collective delivery policy entry
14     lock(collectiveDiscardCondition);              // prohibits race conditions
       // delete entries because policy has been enforced
15     entry = discardCollectivePolicyEntry(msg);
16     for every corresponding entry in the peer delivery policy data structure do
17         discardPeerPolicyEntry(entry);
18     end
19     unlock(collectiveDiscardCondition);
20     return entry;
21 end

22 Function checkCollectiveDeliveryPolicy is
       input  : Message (msg)
       output: true if delivery policy has been enforced, false otherwise
23     policy = getCollectiveDeliveryPolicy(msg.content, msg.sender);
24     if policy == null then
25         return false;                    /* policy has already been enforced */
26     end
27     if msg.subtyp == ACKNOWLEDGE then
28         return policy.checkAcknowledge();     /* returns true if this message
           enforced the policy */
29     else
30         return policy.checkError();  /* can throw an exception or just return
           false */
31 end
```

Algorithm 2.6: Enforcing a delivery policy on the collective level.

3 Experimental Performance Evaluation

The following performance evaluation was made on a machine with the following specifications: Windows 7 64-bit, Intel Core2 Duo with 2x 2.53 GHz, 4.00 GB DDR2-RAM. The simulation configuration is as follows:

- One implementation of a Stateless Output Adapter (one instance shared by all peers).
- 10 Input Push Adapter to receive input from peers.
- Output and Input Adapters communicate directly using a in-memory queue to simulate a peer with a response time of zero.
- Worker threads (workers) simulate the number of applications/users that send messages to the system.
- One million messages are sent for each evaluation test run to get a meaningful average number of messages sent/received.
- Only sent and received messages are considered as 'handled', no internal messages.

Figure 20 depicts the setup for the performance evaluation as described above.

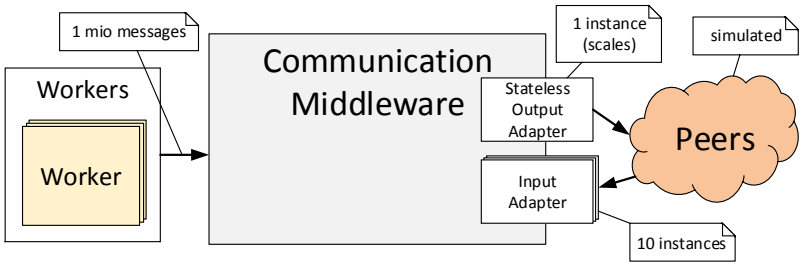

Figure 20: Setup for the performance evaluations.

The performance has been evaluated for every combination of 1, 5, 10, 20, 50 and 100 worker threads (Worker) simulating SmartSociety platform applications sending $1 \cdot 10^6$ messages concurrently, uniformly distributed to 1, 10, 100, and 1000 peers waiting for messages and replying to them. Each test run has been executed 10 times to obtain average throughput results. Figure 21 presents the results of the test runs. The test runs can be reproduced using the stated setup data to configure the Java application located at

GitHub[4]. As one can see, the average throughput remains between 5000 and 3000 messages per second. The performance decrease with higher amounts of peers is result of increased memory requirements rather than computational complexity. The limiting factor here is the used ActiveMQ message broker which only allows a maximum of approximately 20000 messages per second. The system has an upper bound of 5000 messages per second since each message is handled multiple times by the message broker and the SmartCom. This limitation applies to a single SmartCom instance, but multiple SmartCom instances can be deployed to balance the load if needed, sharing the database and PeerManager access. The chosen numbers of worker threads and peers cover the maximum number of concurrent SmartSociety platform applications and collective members, respectively, expected to use a single SmartCom instance. Performance is not expected to become a primary concern of SmartCom due to the increased latency of human peers and variance of response times compared to machine peers.

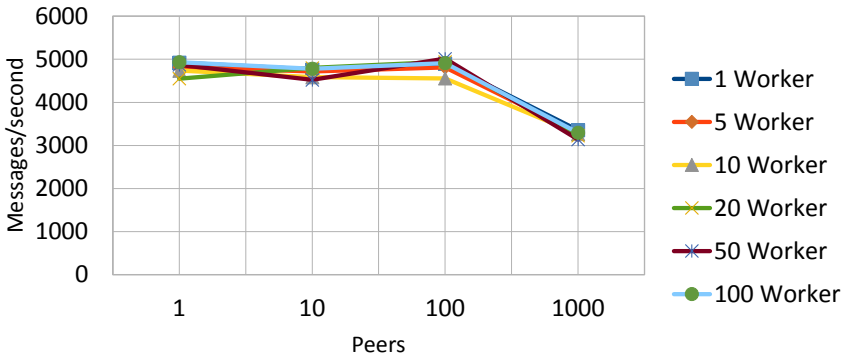

Figure 21: Measured message throughput. Workers simulate SmartSociety platform applications sending out messages to the peers.

References

[1] A. Fipa, "Fipa acl message structure specification," *Foundation for Intelligent Physical Agents, http://www. fipa. org/specs/fipa00061/SC00061G. html (30.6. 2004)*, 2002.

[4] `https://github.com/tuwiendsg/SmartCom/blob/master/smartcom-demo/src/main/java/at/ac/tuwien/dsg/smartcom/demo/PerformanceDemo.java`