SmartCom Communication & Virtualization Middleware Architecture v2.2

Authors:
Philipp Zeppezauer (TUW),
Ognjen Scekic (TUW),
Mirela Riveni (TUW),
Hong-Linh Truong (TUW)

Table of Contents

- Introduction
- Functional Requirements and Relation to other WP Components
- Messages
  - Content of Messages
  - Routing of Messages
  - Predefined Messages
- Architecture
  - Overall System
  - Components
    - Authentication Manager (AuthM)
    - Messaging and Routing Manager (MaRM)
      - Routing
    - Adapter Manager (AM)
      - Adapters
    - Message Query Service (MQS)
    - Message Info Service (MIS)
  - Putting it all together
- Application Programming Interfaces (APIs)
  - Public Entities
  - Callback Entities
  - Communication API
    - send
    - addRouting
    - removeRouting
    - addPushAdapter

        - addPullAdapter
        - removeInputAdapter
        - registerOutputAdapter
        - removeOutputAdapter
    - Output Adapter API
        - push
    - Input Push Adapter API
        - publishMessage
        - init
        - preDestroy
    - Input Pull Adapter API
        - pull
    - Message Info Service API
        - GetInfoForMessage
        - addMessageInfo
    - Message Query Service API
        - query
    - PMCallback API
        - getPeerAddress
        - authenticate
    - Notification Callback API
        - notify
    - Collective Info API
        - resolveCollective
- Internal Functioning and Interaction with SmartSociety Platform Components
    - Motivating scenario
        - General scenario
        - Use-case
    - Creation of output adapters
    - Creation of input adapters
    - Send messages to peers and collectives
    - Receive input

Introduction

The **SmartSociety Communication Middleware** (hereafter: **Middleware**, WP7) is the component providing the communication and peer virtualization functionality to other SmartSociety components, as defined in T7.1.

The Middleware provides suitable communication functionality between the SmartSociety Platform components executing a SmartSociety program on one side, and peers (ICUs - Individual Computing Units), i.e., human-based services (workers, HCUs) and software-based services (machines, MCUs) on the other side. More specifically, the Middleware allows exchanging of control and data messages between different components connecting to the Middleware, running both at Platform side and on peer[1] side. This effectively allows the virtualization of peers to the rest of the Platform, with respect to the following phases of collective's lifecycle:

1. Locating (matching) of appropriate peers
2. Assembling and provisioning a collective for a given task, including:
    a. Selecting the optimal composition of the collective
    b. Task acceptance negotiations
    c. Imposing a structure/communication/management topology
    d. Enacting a collaborative pattern/workflow
3. Monitoring of task execution on peers
4. Advertising of incentives to the peers
5. Collecting subtask results
6. Finalizing the task execution

The Middleware provides low-level communication and control primitives through which each of the above-listed functionalities can be implemented, thus effectively virtualizing the peers to the rest of the Platform. Middleware offers the asynchronous (message-based) communication functionality for interacting with the peers. This communication functionality comprises:

a. message transformation
b. message routing
c. message delivery with configurable delivery options
   (e.g., retries, expiry, delayed, acknowledged)
d. message persistence
e. message querying
f. message interpretation service for peers

[1] We occasionally use the encompassing term ICU to denote an individual software or human-based peer providing a service within a context of a SmartSociety task execution.

The Middleware does not provide an environment for peers to actually execute the task, nor does it strive to be the only or preferred communication channel for peers participating in the task execution. Rather, the peers should be encouraged to self-organize to work and collaborate using the well-known tools they feel comfortable with (e.g., email, Dropbox, Google maps, Skype) whenever possible. Decoupling from particular collaboration and communication tools also allows us to support a wide variety of labor processes. This enables the *smart*-ness of SmartSociety, because it leaves the liberty to the participating peers of organizing the execution, communication and coordination whenever possible, as they would do in the real world. The SmartSociety should be there to set the borders of this liberty, and to provide mechanisms to set up, monitor and finalize the execution, while guaranteeing to the participating peers a fair and rewarding environment. This design philosophy guides the design of the Middleware component.

Functional Requirements and Relation to other WP Components

Figure 1 shows the conceptual architecture of the Middleware. The particular components are described in more detail in the following sections.

The communication with peers has to be handled independently of their actual type, allowing for a uniform communication with peers. This is achieved by using different adapters for different peer types. The implementation of those adapters is left to the programmer of applications for the SmartSociety Platform. The Middleware will provide some basic, commonly used adapters.

Using the Middleware the *Task Execution Engine*[2] (WP6,WP7) and other SmartSociety Platform components should be able to send messages to peers which are identified by a unique identifier (peer ID). The mapping of the peer ID and the actual address/communication channel will be handled by the Middleware with the assistance of the *Peer Manager* (WP4) and the *Elastic Provisioning Engine*[3] (WP7,WP6). The *Elastic Provisioning Engine*, while functionally not part of the Middleware is tightly related to it, as it provides the fundamental functionality of managing the collective composition and lifecycle. Logically, the *Elastic Provisioning Engine* can be seen as part of the *Task Execution Engine*, as assembling a collective can be seen as the initial lifecycle phase of a task execution. Since *Elastic Provisioning Engine* also makes use of peer metrics and profiles, it will need to closely relate to the *Peer Manager* (WP4) component. Note that there might be multiple mappings of peer IDs to addresses/communication channels (e.g., a peer can be contacted using a mobile application and/or sms, or access the system via a mobile app and web browser concurrently).

The Middleware supports unicast as well as multicast messages based on peer IDs. Therefore peers can be addressed as collectives and the Middleware will take care of sending the message to every single member of the collective. This behavior will be provided by the *Messaging and Routing Manager* who checks with the Elastic Provisioning Engine which peers belong to the collective.

Note that the communication between peers and the adapters is unidirectional -- output adapters are used to send messages to the peers; while input adapters are used to receive messages from peers. The reason behind this behavior is not to restrict the request and the response on using the same communication channel, but also to be able to filter the incoming messages from peers and support transformations of message formats not known at design time.

[2] The Task Execution Engine itself is not part of the Middleware.

[3] At the time of writing, the Elastic Provisioning Engine manages collectives following WP7's HCU models. However, at the later stage this functionality will be extended/replaced by components produced by other WPs, mostly WP6. The Provisioning Engine itself is not part of the Middleware.

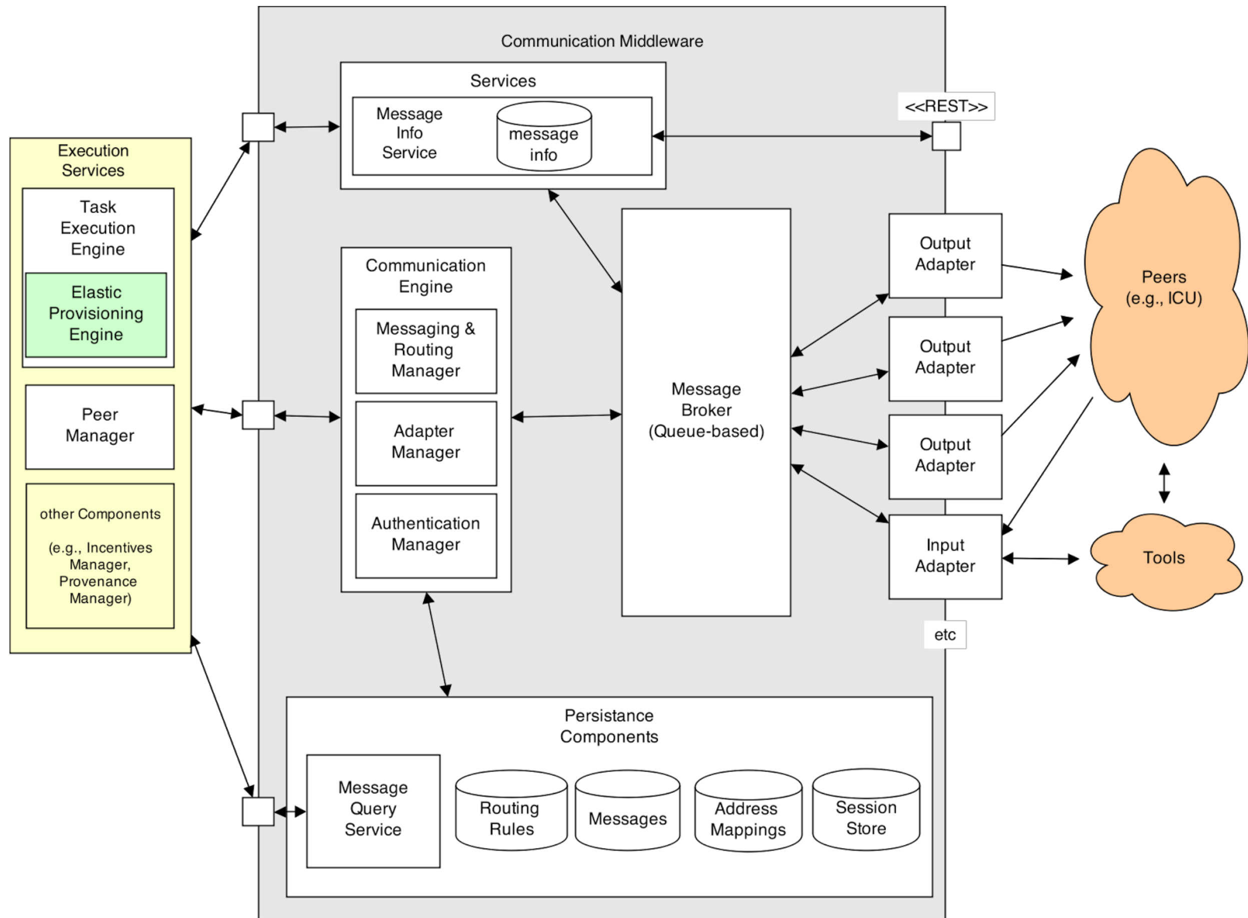

Figure 1: Conceptual architecture of the Middleware

Input messages from peers (e.g., subtask result) or external tools (e.g., dropbox file added, email received on a mailing list) will be handled by the Middleware either by push or by pull in regular intervals. Therefore the Middleware maintains special input adapters and executes them appropriately (e.g., perform a pull every few seconds).

The input and control messages will be routed by the Messaging and Routing Manager to the corresponding Middleware-internal component or to the Task Execution Engine. This behavior depends on the routing rules that will be handled by the Middleware and can be expanded by the Task Execution Engine.

All communication between the peers and the platform will be handled asynchronously using messages. The Middleware will use a queue-based message broker to decouple the execution of the platform and the communication. Furthermore the usage of output adapters and input adapters support the asynchronous intention of the system.

To provide security functionality the peers have to be authenticated before interacting with the system. This information will be provided to/from the *Peer Manager* (WP4). The *Authentication Manager* will also create a security token for the peer that can be used to authenticate the messages.

All sent and received messages as well as internal messages will be persisted. Stored messages can be queried and analysed to derive metrics/incentives or to gain further insights in what is going on in the Middleware.

Besides the requirements that relate to the communication of the platform with peers there are a few other requirements that relate to supporting components and services:

An info service should provide peers with the semantic meaning of messages and how they are related to other messages based on their type and subtype.

A service that provides a query service for exchanged messages between the platform and peers as well as internal messages.

The info service should be accessible through a REST API.

Messages

The Middleware exchanges messages with the Task Execution Manager, the output adapters as well as with some internal components. The following chapter will take a look at how those messages will look like and discuss some basic message types.
Note that further message subtypes can be defined by programmers of applications for the SmartSociety Platform.

Content of Messages

Fields that are marked with (o) are optional.

Field	Description
Type	This field defines the low-level purpose of the message (e.g., contro message, input message, metrics message,...). This field is especially important for the routing of messages within the system.
Subtype	The subtype of a message is defined by the component that is in charge of the message (component specific). The subtype combined with the type of the message define the actual purpose of the message. The subtype can also be used by programmers of applications for the SmartSociety Platform to define custom message types for their application.
Message-Id	Every message within the system will be assigned a global unique identifier by the Messaging and Routing Manager.
Sender-id	The sender-id specifies the sender of the message (can be a component, peer, …). Sender-ids are unique within the systems. Sender-IDs are either predefined (in case of platform components like the Task Execution Engine) or will be assigned by the Pee Manager.
Receiver-id (o)	The receiver-id specifies the receiver of the message (can be a component, peer, collective). Note that the receiver-id can also be empty, in that case the routing is solely based on the type and subtype of the message but might fail because there are no routes available. The receiver-id can also be a collective of peers, in tha case the Middleware will determine the actual recipients of the message and will perform the multicast.
Conversation-Identifier (o)	Denotes the system identifier for the conversation (workflow). This identifier will be used by the Task Execution Engine to map the message to the actual execution instance of an application.

	For example: program A will be executed twice at the same time: A1 and A2. The conversation-id is then used to match the message to the to execution A1 or A2. If there is no such conversation (e.g., fo internal Middleware messages), the conversation-id can 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 and a response will be handled by the system (if there is such a response). For example: a peer has one hour to post pictures in a dropbox folder, after this time the Middleware will stop looking for pictures in the folder and will create an error message if there are no pictures.
Language (o)	Denotes the language of the system. This can be a natural language like English or German, as well as a computer format like binary. In the first stage this field has no intended usage besides logging and debugging purpose. In future stages a translation service could be introduced that uses this field.
Security-Token (o)	A security token that can be used to guarantee the authenticity of messages or to encrypt the content of the message.

Note that some Middleware components might add additional data to the message. After receiving a message, output adapters are responsible to transform them to the appropriate peer-understandable representation and send the messages. On the other hand, input adapters are responsible for the transformation of messages of a different message format (e.g., email) to an internal message.

Routing of Messages

The routing of messages is handled by the Messaging and Routing Manager and according to rules based on the type, subtype and receiver-id. Note that the order (type, subtype, receiver-id) also defines the priority, which means that the type has the highest priority and the receiver-id the lowest (because it can be empty for some internal message).

Predefined Messages

Message	Description
Acknowledge	This message will be 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 will not be sent if the programmer requires a fire-and-forget sending behavior (i.e., she doesn't care if it actually has been delivered).
MetricsRequest	This message is sent from peers to the Task Execution Engine to request the personal metrics.
MetricsResponse	This message is sent from the Task Execution Engine to the peer, containing the previously requested peer metrics.
MessageInfoRequest	Request by a peer to the Message Info Service for information o how to interpret and handle a given message based on its type subtype and conversation-id.
MessageInfoResponse	Response of the Message Info Service to a peer that contains information on how to interpret and handle a given message base on its type, subtype and conversation-id.
Authenticate	Authentication message of a peer that contains its credentials.
AuthenticateResponse	Response message for a authenticate message from the Middleware to the peer. Might contain security information if there ar some.
MonitoringInfo	Message type indicating that the message load contains the metri update (value), that will be statically routed to the Task Executio Engine.
Error	A generic error message that indicates that there has been an error This message will be handled based on the routing rules.
CommError	This error message indicates that there has been an error during the communication. This will be reported to the Task Execution Engine which is responsible for the handling of such errors.
Timeout	This message indicates that a timeout has appeared in the system and that the message couldn't be delivered in time or there was n response within a certain time.

Architecture

We will use the following convention in the diagrams:

- Yellow components indicate components that are part of the SmartSociety Platform but not of the Middleware itself, OR functionally belong to the Middleware, but have to be implemented by the programmer of an application for the platform (e.g.,.application-specific adapters)
- Blue components indicate APIs that are publically available
- Green components indicate components that are formally not part of the Middleware but will be shipped with it to provide certain functionality.
- Grey boxes serve to visually group and improve the visibility of related components.
- Orange components are external to the platform and the Middleware (e.g., peers and their communication channels).
- Arrows indicate the information flow within the system.
- Dotted lines with arrows indicate the calling direction of the middleware and the external components.
- Small white boxes indicate interfaces and communication beyond the boundaries of the components.

We will use the following convention in the description of the components:

- Italic expressions are the names of the components, queues etc. of the Middleware or the platform.

Overall System

See Figure 1 for the overall system.

Components

Authentication Manager (AuthM)

The *Authentication Manager* will be used to authenticate peers in the system. Authentication request messages will be dropped in the *AUTH Queue* and collected by the *Authentication Request Handler*. The handler will contact the Peer Manager (WP4) using the *PMCallback* (which acts as a placeholder for the actual implementation) 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 the Middleware to provide security features (e.g., message authentication or message encryption). This token is only valid a predefined period of time, the time period from the creation of the token and the invalidating of the token will be called *Session*. The *Authentication Provider* can be used by the *Messaging and Routing Manager* to check the authenticity of a message.
The *Authentication Manager* will use a *Session Store* to handle the sessions of peers.

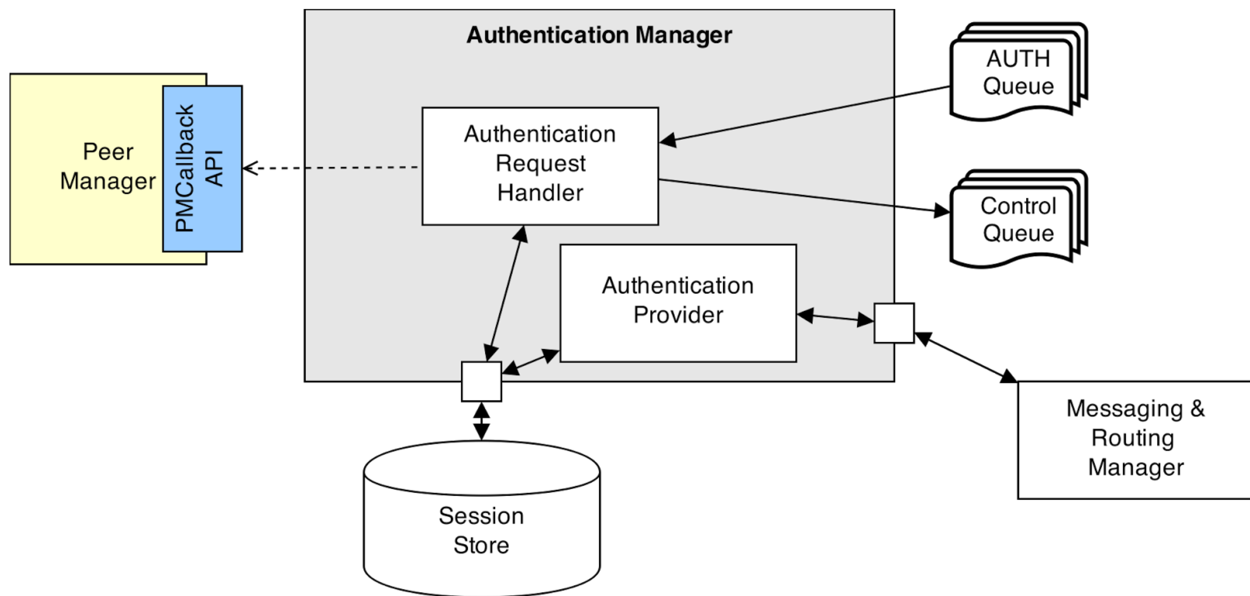

Figure 2: Authentication Manager

Messaging and Routing Manager (MaRM)

The *Messaging and Routing Manager* is responsible for the handling of messages within the system.
The *Task Execution Engine* can issue messages that will be sent using the *Message Handler* to peers or collectives. Upon receiving a message the *Message Handler* contacts the *Routing Rule Engine* to determine the routing rule and puts the message in the corresponding queue or passes it to the *Task Execution Engine* (e.g., a response).
The *Routing Rule Engine* decides the destination of a message based on *Routing Rules*. There will be a predefined set of rules for control messages and other special messages. Further rules can be added by the *Task Execution Engine* to simplify the communication and reduce overhead (e.g., in an application a message of a specific subtype will always be transferred to a software service).
The *Input Handler* will listen to the *Input Queue* for incoming input messages (e.g., a response from a peer) and to the *Control Queue* for incoming control messages (e.g., a communication error message). Those messages will be given to the *Message Handler* that determines what to do with it (usually sending it to the *Task Execution Engine* in the case of a input message). 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* will be responsible of logging all sent and received messages to a persistent database. Furthermore, it is possible to query the history of the messages (see Message Query Service).

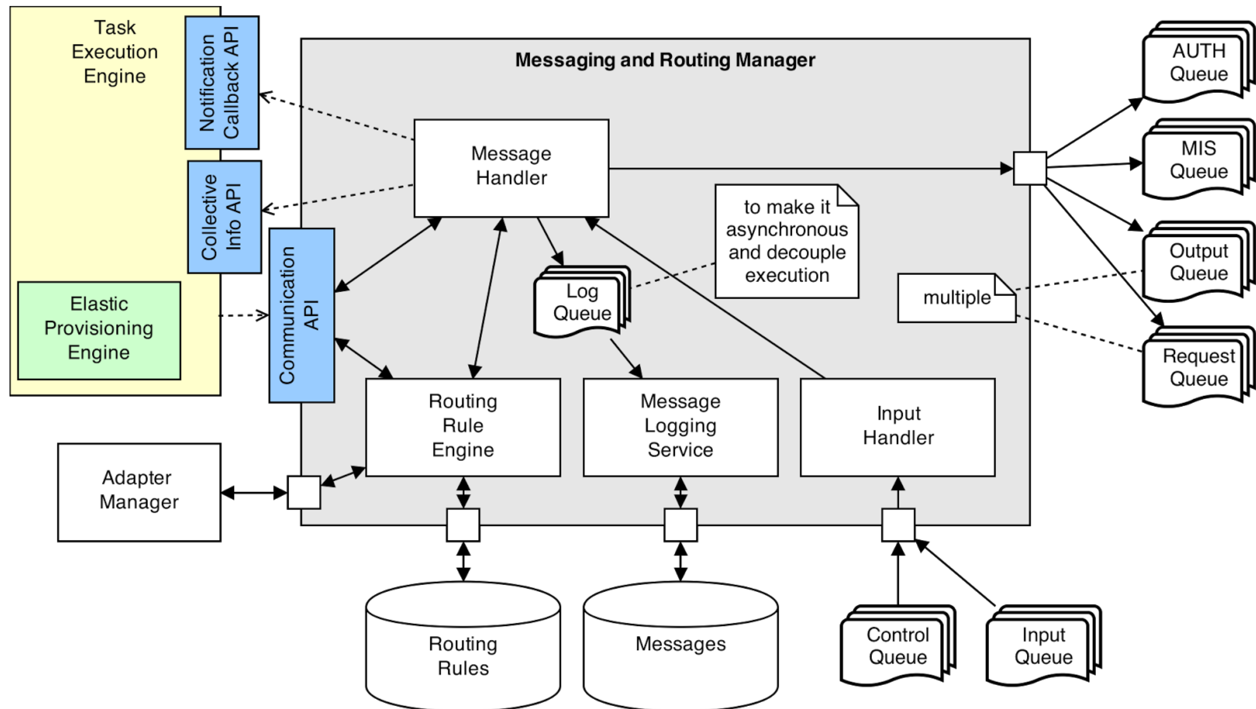

Figure 3: Messaging and Routing Manager

Routing

Routing in SmartCom is handled based on the message's receiver, type and subtype. The resulting routing defines the recipient of the message which can either be a specific output adapter instance, an internal component (e.g., Authentication Manager) or the Task Execution Engine (i.e., usually just input).

As mentioned before, the routing rule is determined using the receiver, type and subtype fields of a message. Note that each of them can be null but it is not allowed that all of them are null, in that case the message will be discarded because the system doesn't know what to do with it.

The route is determined by the properties in ascending order. First the receiver will be determined, afterwards the type and finally the subtype. The Routing Rule Engine uses a technique called Pipes-and-Filters for the routing. Each step (one for each property) returns a list of identifiers which can either be a components or a output adapter instance. Further steps will receive the list of identifiers of the previous step, determine the route, calculate an intersection based on the own results on the one of previous steps and forward the resulting list to the next step. If one property is empty or null, the step will be skipped. In the following we will discuss the steps in detail:

- **Resolving route based on receiver** – the Routing Rule Engine maintains a list of mappings from the receiver ID to an Identifier. Note that there might be multiple Identifiers per receiver ID because a peer could be contacted using multiple adapters.

E.g.: a message has to be sent to peer A. There are mappings available for an SMS adapter, an Email adapter and a Skype adapter. The Routing Rule Engine will then return a list with all found identifiers (SMS, Email and Skype adapters in our case).
It is possible that the receiver is set but there is no route available. There are two ways to handle to situation:

  - **receiver is not a peer identifier** – if the message has been sent by the Task Execution Engine, the Routing Rule Engine will report back that it can't handle the message. Otherwise an error will be logged. Note that this case usually should not happen because there are predefined routes for every internal component.
  - **receiver is a peer identifier** – the Routing Rule Engine will instruct the Output Adapter to create a new adapter for the given receiver. After the creation of such an adapter, the Routing Rule Engine will create a new mapping from the peer identifier to the newly created adapter instance.

- **Resolving route based on type** – the Routing Rule Engine maintains a list of mappings from message types to an Identifier. Based on the type it will determine zero, one or multiple identifiers.
- **Resolving route based on subtype** – the Routing Rule Engine maintains a list of mappings from message subtypes to an Identifier. As in the previous step, it will determine the identifiers.

If the resulting list contains more than one identifier, it's up to the caller of the Routing Rule Engine to determine the optimal receiver of a message. This is especially important if there is a preferred adapter (e.g., mobile application) that should be used for communication. If the resulting list is empty, the route should only be determined based on the message's receiver - if set.

Adapter Manager (AM)

The *Adapter Manager* is responsible for the initialization and lifecycle management of communication adapters for peers and tools (e.g., Dropbox). More specifically, the *Adapter Execution Engine* initializes and executes the adapters (e.g., performs a pull request after a certain interval, sends messages to peers) and is able to scale out adapters to handle big workloads. Note that there might be *Stateful Output Adapters* and *Stateless Output Adapters*, which differ in behavior (see Adapters).
The *Task Execution Engine* is able to create *Input Adapters* using the *Communicaton* API, that will receive input from external tools or peers. The *Task Execution Engine* is not able to create output adapters.
The *Messaging and Routing Manager* is able to create output adapters to contact peers if no such adapter exists. The corresponding adapter address will be returned to the handler.
If the *Adapter Handler* has to create a new Output Adapter, it might contact the *Peer Manager* to determine the address and the method of how to contact a peer. Address information can be stored in the *Address Mappings* Database using the *Address Resolver* which is also responsible to resolve persisted address requests by adapter executions.
The *Address Mappings* database stores contact information of peers for adapters. It acts as a cache for information that can be retrieved from the *Peer Manager* using the *PMCallback*. It

stores information like the Skype name of a peer that is needed by a Skype adapter to send messages to the peers Skype account. This information will be resolved by the *Adapter Execution* and passed to the adapter.

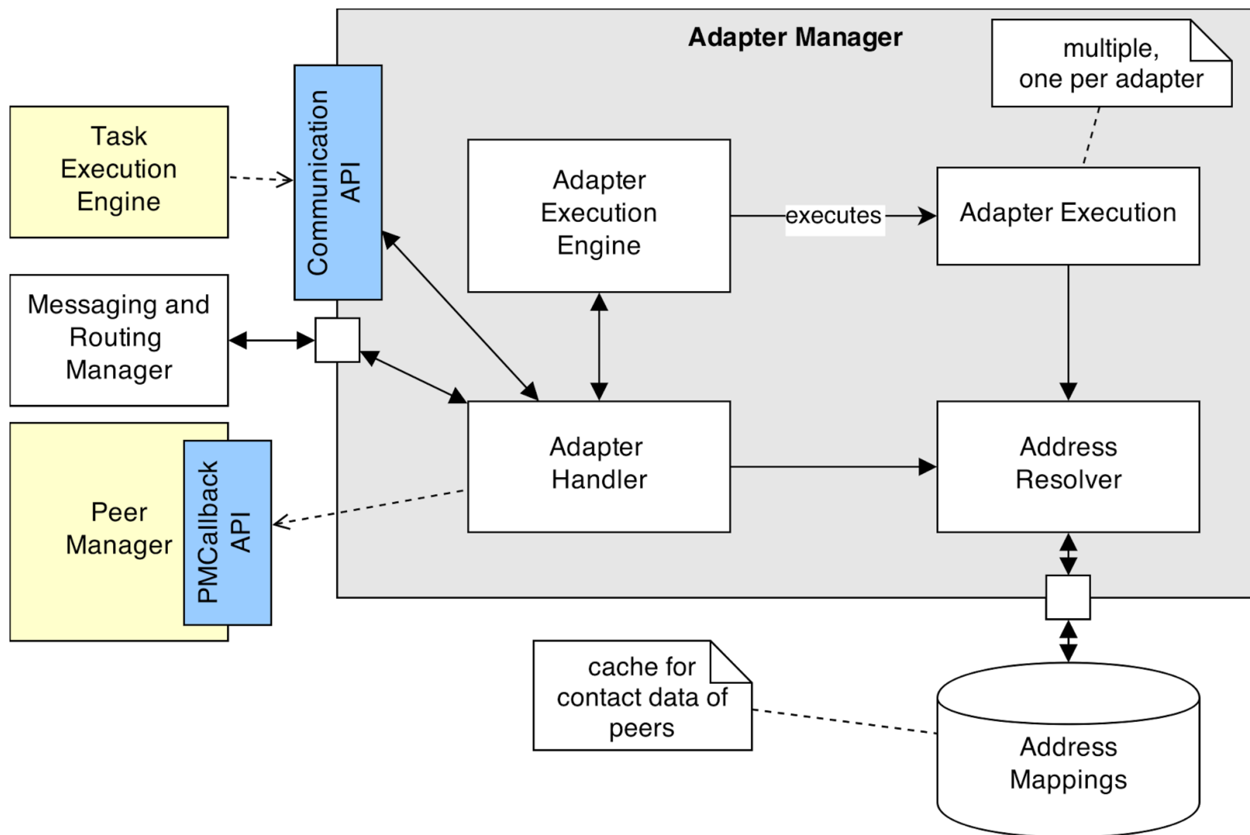

Figure 4: Adapter Manager

Adapters

The Middleware uses adapters to abstract and virtualize peers to the rest of the platform. There are two different type of adapters: *Output Adapters (PA)* and *Input Adapters (FA)*. They differ in the behavior as the *Output Adapter* is only allowed to contact a peer and the *Input Adapter* is only allowed to receive input from an external tool or peer.

Furthermore there are two categories of *Output Adapters*: *Stateful Output Adapters (SFPA)* and *Stateless Output Adapters (SLPA)*. *Stateful Output Adapters* are only used for one specific peer whereas *Stateless Output Adapters* can be used for multiple peers at the same time. The necessary data to contact a peer with a *Stateless Output Adapter* will be provided by the *Adapter Manager* (e.g., a telephone number). *Stateful Output Adapters* are responsible to store by themselves how to contact a peer (they will be provided with that data at creation).

Input Adapters can be implemented using push or pull mechanisms. Push adapters will be notified by the external tool/communication channel of new developments (e.g., a new mail in the mailinglist). Pull adapters will be handled by the *Adapter Execution Engine* and the pull will be

triggered in a certain interval or based on a programmed request (e.g., a peer has only 1 hour to send a file to dropbox, after this hour, the pull adapter will check if there is such a file).
The output adapter also has a control queue, which will be used for the issuing of metrics and special control messages (e.g., acknowledgement messages).

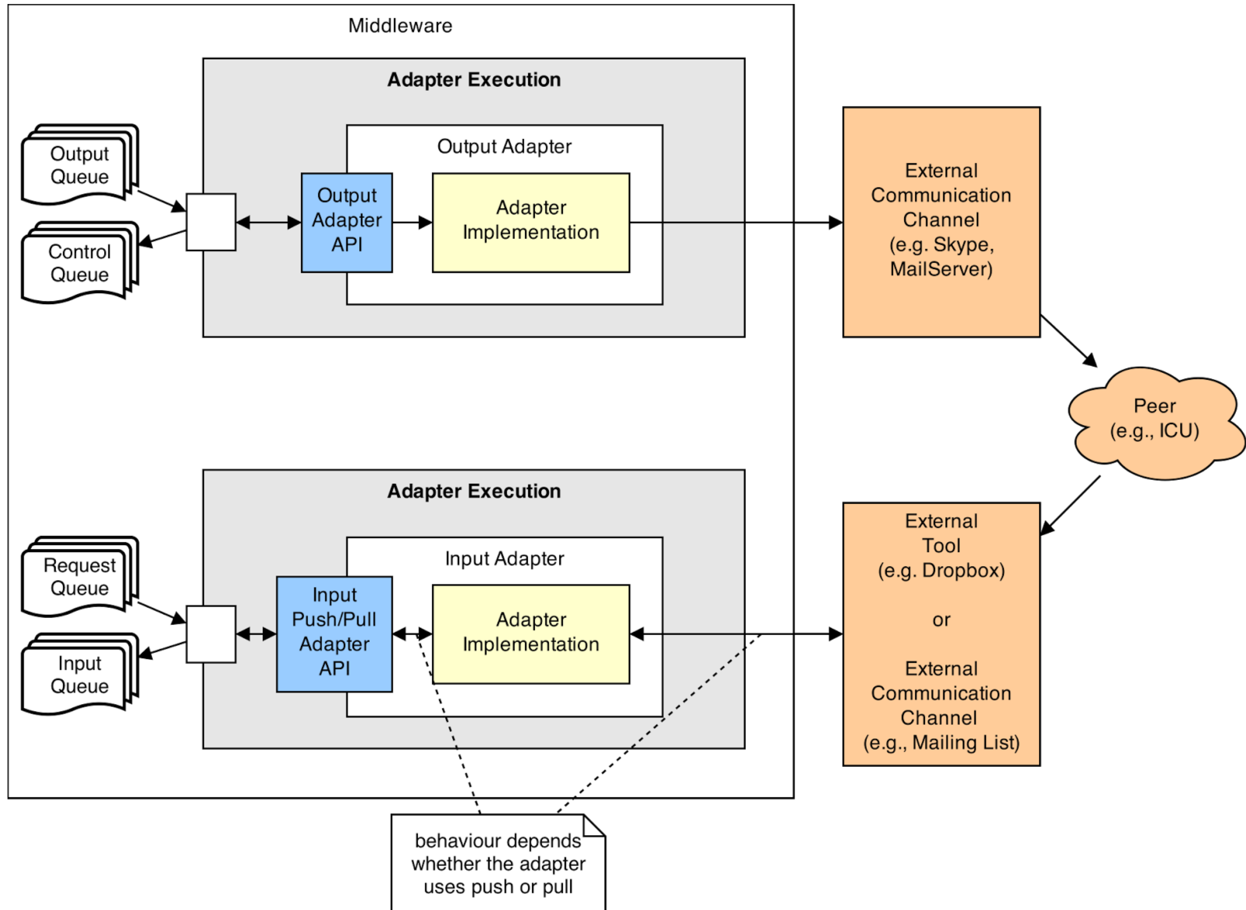

Figure 5: Output Adapter and Input Adapter

Message Query Service (MQS)

The *Message Query Service* provides an interface to query sent and received messages.

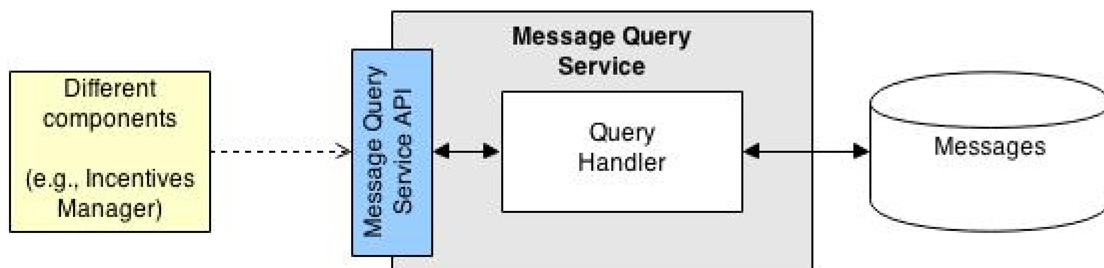

Figure 6: Message Query Service

Message Info Service (MIS)

The *Message Info Service* provides information on messages based on their type and subtype. It will be used by peers to get information on how to interpret a message and how to respond. Furthermore it will provide a human-readable description of the message's structure and contents, as well as its semantic meaning and relation with other messages. At a later stage this service can be improved to return an ontology explaining how to interpret the message in a machine-readable way; at this at this stage it is enough to provide a simple textual description that the worker can fetch to interpret the message semantics, especially with respect to related messages. The service maintains a database to store the message information which will be updated through the *Task Execution Engine*.

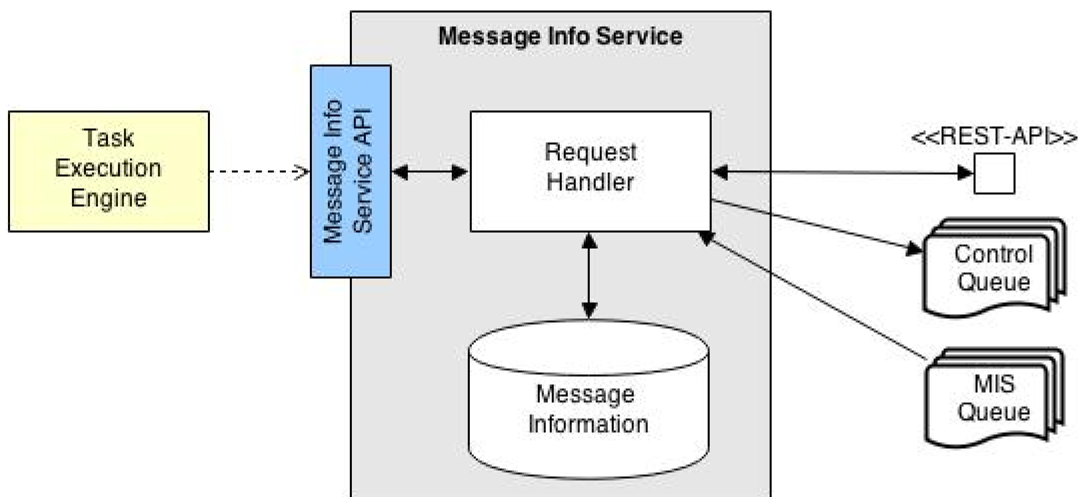

Figure 9: Message Info Service

Putting it all together

The blue boxes are APIs that will be used to interact with the system. Those APIs will be described in the following chapter.

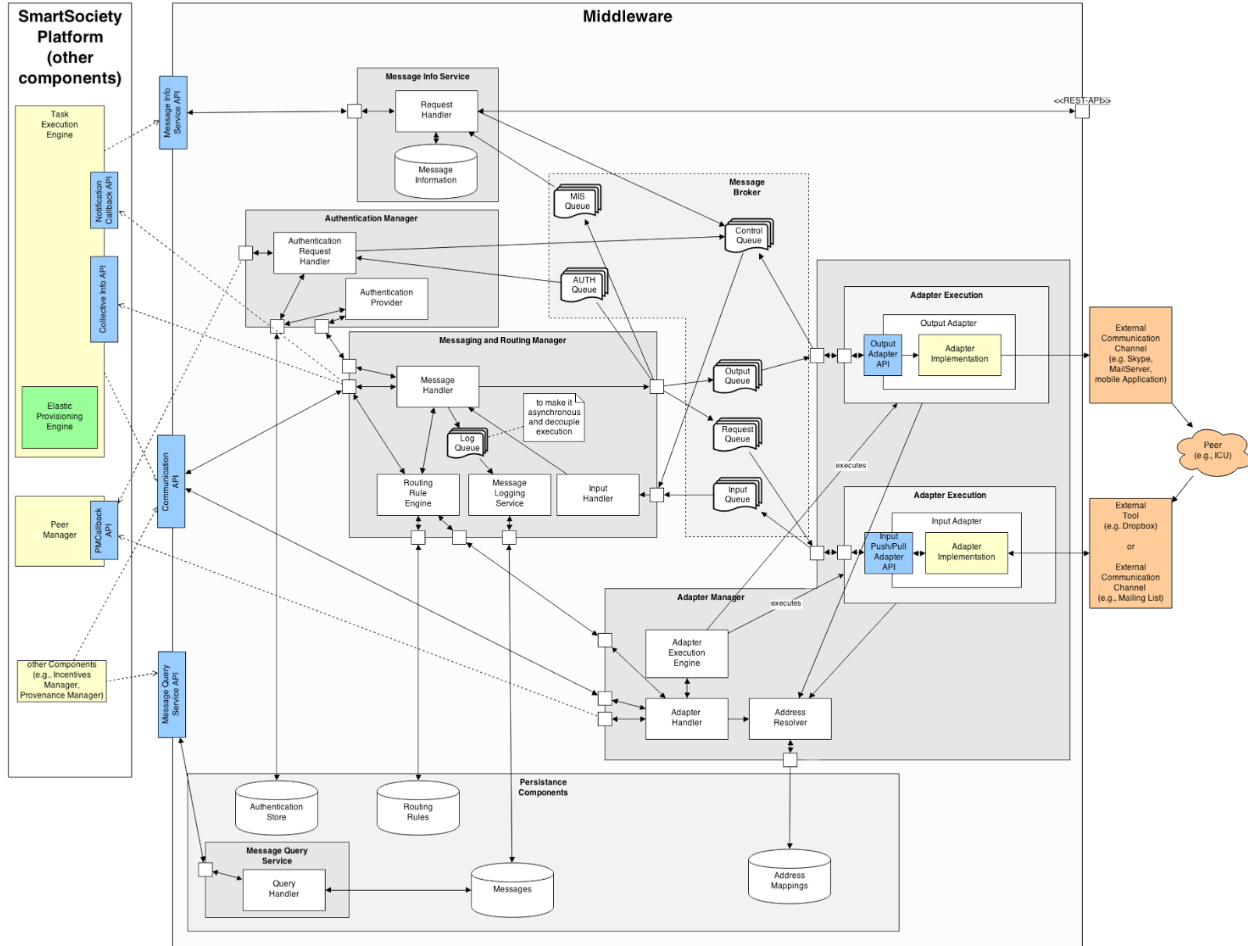

Figure 10: Detailed components of the middleware

Application Programming Interfaces (APIs)

Public Entities

Entity	Description
Communication	Main entity that will be used for the communication with the system.
Message	Message that will be exchanged with the communication class.
MessageInfoService	Service that provides information on messages. Can be used to ask for information on messages or add message information to the service.
MessageInformation	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.
QueryCriteria	Criterias for querying messages in the MessageQueryService
Profile	Profile that relates to a specific peer.
OutputAdapter	Adapter that sends messages to the peers. New adapters have to implement the send method.
InputPushAdapter	Adapter that receives messages from peers or tools by push. New adapters have to implement the pull method.
InputPullAdapter	Adapter that pulls messages from peers or tools. New adapters have to call the publishMessage method to announce to the system that a new message has been received.
RoutingRule	Defines a routing rule that can be defined by the Task Execution Engine to simplify the communication and reduce overhead (e.g. messages from peer A with type T are always sent to peer B).

Callback Entities

Those entities will be used by the system to interact with SmartSociety components that are not part of the Middleware but that Middleware depends upon and communicates with.

Entity	Description
PMCallback	Callback for the Peer Manager.
CommunicationCallback	Callback for the Task Execution Engine.
NotificationCallback	Callback that will be notified if new input messages arrive.

Communication API

This section will discuss the main API for the SmartSociety Platform to interact with peers and the middleware for the purpose of communication. It provides methods to start the interaction with collectives and single peers and also defines methods to extend and manipulate the behaviour of the middleware.

Method	Description
send	Sends a message. This message is sent to a collective or a single peer. The method returns after the peer(s) have been determined. Errors and exceptions thereafter will be sent to the Notification Callback. Optional receipt acknowledgements are communicated back through the Notification Callback API.
addRouting	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).
removeRouting	Remove a previously defined routing rule identified by an ID.
addPushAdapter	Creates a input adapter that will wait for push notifications. Returns the ID of the adapter.
addPullAdapter	Creates a input adapter that will pull for updates in a certain time interval. Returns the ID of the adapter. The pull requests will be issued in the specified interval until the adapter is explicitly removed from the system.
removeInputAdapter	Removes a input adapter from the execution.
registerOutputAdapter	Registers a new type of output adapter that can be used by the middleware to get in contact with a peer. The output adapters will be instantiated by the middleware on demand.
removeOutputAdapter	Removes a type of output adapters. Adapters that are currently in use will be removed as soon as possible (i.e., current

	communication won't be aborted and waiting messages in the adapter queue will be transmitted).

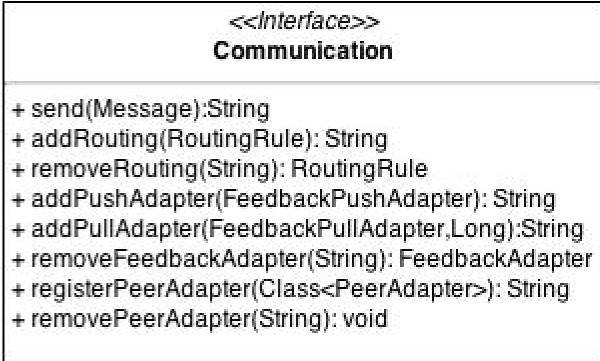

Figure 11: Communication API

send

Sends a message. This message is sent to a collective or a single peer. The method returns after the peer(s) have been determined. Errors and exceptions thereafter will be sent to the Notification Callback. Optional receipt acknowledgements are communicated back through the Notification Callback API.

Return

Returns the internal ID of the middleware to track the message within the system.

Parameters

Parameter	Type	Description
message	Message	Specifies the message that should be handled by the middleware.

Throws

CommunicationException - a generic exception that will be thrown if something went wrong in the initial handling of the message.

addRouting

Add a special 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). The middleware will check if the rule is valid and throw an exception otherwise.

Return

Returns the middleware internal ID of the rule

Parameters

Parameter	Type	Description
rule	RoutingRule	Specifies the routing rule that should be added to the routing rules of the middleware.

Throws

InvalidRuleException - if the routing rule is not valid.

removeRouting

Remove a previously defined routing rule identified by an ID.

Return

The removed routing rule or nothing if there is no such rule in the system.

Parameters

Parameter	Type	Description
ruleId	String	The ID of the routing rule that should be removed.

Throws

nothing

addPushAdapter

Creates a input adapter that will wait for push notifications or will pull for updates in a certain time interval. Returns the ID of the adapter.

Return

Returns the middleware internal ID of the adapter.

Parameters

Parameter	Type	Description
adapter	InputPushAdapter	Specifies the input push adapter.

Throws
nothing

addPullAdapter

Creates a input adapter that will pull for updates in a certain time interval. Returns the ID of th adapter. The pull requests will be issued in the specified interval until the adapter is explicitly removed from the system.

Return
Returns the middleware internal ID of the adapter.

Parameters

Parameter	Type	Description
adapter	InputPullAdapter	Specifies the input pull adapter
interval	long	Interval in milliseconds that specifies when to issue pull requests. Can't be zero or negative.

Throws
nothing

removeInputAdapter

Removes a input adapter from the execution.

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

Parameters

Parameter	Type	Description
adapterId	String	The ID of the adapter that should be removed.

Throws
nothing

registerOutputAdapter

Registers a new type of output adapter that can be used by the middleware to get in contact with a peer. The output adapters will be instantiated by the middleware on demand.

Return

Returns the middleware internal ID of the created adapter.

Parameters

Parameter	**Type**	**Description**
adapter	OutputAdapter	The output adapter that can be used to contact peers.

Throws

nothing

removeOutputAdapter

Removes a type of output adapters. Adapters that are currently in use will be removed as soon as possible (i.e., current communication won't be aborted and waiting messages in the adapter queue will be transmitted).

Return

nothing

Parameters

Parameter	**Type**	**Description**
adapterId	String	Specifies the adapter that should be removed.

Throws

nothing

Output Adapter API

The Output Adapter API will be used to implement an adapter that can send (push) messages to a peer. Therefore the push method has to be implemented. Output Adapters will receive a message, transform this message and push it to the peer over an external communication channel (e.g., send the message to a web platform or a mobile application).

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

Figure 12: OutputAdapter API

push

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

Return
nothing

Parameters

Parameter	Type	Description
message	Message	Message that should be sent to the peer.
address	PeerAddress	The address of the peer

Throws
nothing

Input Push Adapter API

The Input Push Adapter API can be used to implement an adapter for a communication channel that uses push to get notified of new messages. The concrete implementation has to use the InputPushAdapterImpl 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 InputPushAdapterImpl class. This method delegates the message to the corresponding queue and subsequently to the correct component of the system. The adapter has to start a handler for the push notification (e.g., a handler that uses long polling) in its init method.

Method	Description
publishMessage	Publish a message that has been received from external tools/peers. This method should only be called when implementing a push service to notify the middleware that there was a new message.
init	Method that can be used to initialize the adapter and other handlers like a push notification handler (if needed)
preDestroy	Notifies the push adapter that it will be destroyed after the method returns. Can be used to clean up and destroy handlers and so forth.

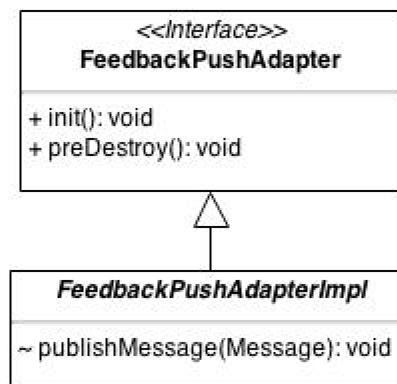

Figure 13: Input Push Adapter API

publishMessage
Publish a message that has been received. this method should only be called when implementing a push service to notify the middleware that there was a new message. Return nothing

Parameters

Parameter	Type	Description
message	Message	Message that has been received.

Throws
nothing

init

Method that can be used to initialize the adapter and other handlers like a push notification handler (if needed)

Return
nothing

Parameters
none

Throws
nothing

preDestroy

Notifies the push adapter that it will be destroyed after the method returns. Can be used to clean up and destroy handlers and so forth.

Return
nothing

Parameters
none

Throws
nothing

Input Pull Adapter API

The Input Pull Adapter is dedicated to pull notifications from external tools or peers. For example it can look at dropbox if there is a new file available. An instance of a input adapter is always assigned to a single task and therefore in the context of the task. It has to transform the notification to an internal message. Upon registration of the input pull adapter in the middleware the interval for pull checks has to be specified. From then on the middleware will call the pull method in the specified time intervals.

Having a stateful pull adapter has some advantages:

- the state of the communication (e.g., the corresponding execution id of input messages) will always be saved in the adapter and there is no need to save it in the adapter manager.
- race conditions due to the parallel execution of an adapter are not possible because each adapter is only executed by a single thread. Therefore no synchronisation 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 at the creation of the adapter and there is no need for a dirty parameter passing to a stateless adapter (e.g., a map of objects/strings).

This approach also has some downsides:

- input pull adapters have to be created in the Task Execution Engine 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.

Method	**Description**
pull	Pull data from a predefined location (only possible if the adapter supports pull). This method has to be implemented by pull adapters.

Figure 14: Input Pull Adapter API

pull
Pull data from a predefined location (only possible if the adapter supports pull). This method has to be implemented by pull adapters. Return Returns a new message or null if there is no new information. Parameters none Throws AdapterException - an exception occurred during the pull operation.

Message Info Service API

The message info service provides information on 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.

Method	Description
getInfoForMessage	Returns information upon the given message. The information contains how to interpret the message and how it's related to other messages (i.e., how to respond).
addMessageInfo	Add information on a given message.

Figure 15: MessageInfoService API

GetInfoForMessage

Returns information on a given message to the caller.

Return

Returns the information for a given message

Parameters

Parameter	Type	Description
message	Message	Can be a valid message ID or an instance of Message.

Throws

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

addMessageInfo

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

Return
nothing

Parameters

Parameter	Type	Description
message	Message	Specifies the type of message.
info	MessageInformation	Information for messages of the type of parameter message.

Throws
nothing

Message Query Service API

This service can be used to query the logged messages. To query the service, a QueryCriteria object has to be used that specifies the query.

Method	Description
query	Queries the message log for messages that match a given query statement (similar to the JPA Criteria API[4]).

Figure 16: MessageQueryService

query

Queries the message log for messages that match a given query statement (similar to the JPA Criteria API).

Return

Returns a collection of messages that match the given query statement.

Parameters

Parameter	Type	Description
statement	QueryCritera	Statement that will be used to query the message log using the QueryCriteria class (similar to the JPA Criteria API).

Throws

IllegalQueryException - if the query is not valid.

[4] http://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

PMCallback API

The Peer Manager Callback (PMCallback) will be used to ask the Peer Manager for different addressing possibilities for a specific peer (e.g., phone number, email address, skype username) and an adapter will be selected based on the returned address. There might be multiple ways on communication with a single peers, therefore the method returns a collection of addresses.

This API is also used to check the user credentials with the PM. This allows the Middleware to issue authentication tokens that will get embedded within the messages exchanged subsequently between peer applications and output/input adapters and enable message authentication, and possibly encryption.

Method	Description
getPeerAddress	Resolves the address(s) of a given peer (i.e, provides the address and the method/adapter that should be used).
authenticate	Authenticates a peer, i.e. checks with the PM if the provided credentials match the peers credentials in the system.

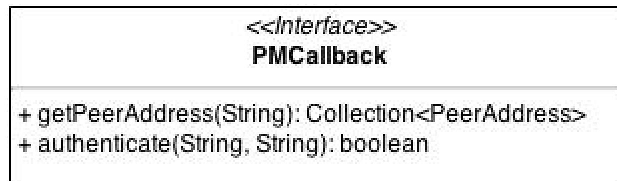

Figure 17: PMCallback interface

getPeerAddress

Resolves the address(s) of a given peer (i.e, provides the address and the method/adapter that should be used).

Return

Returns a collection of all addresses and methods/adapters which can be used to contact the peer.

Parameters

Parameter	Type	Description
peerId	String	id of the requested peer

Throws
NoSuchPeerException - if there exists no such peer.

authenticate

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

Return
Returns true if the credentials are valid, false otherwise

Parameters

Parameter	Type	Description
peerId	String	id of the peer
password	String	password of the peer

Throws
PeerAuthenticationException - if an authentication error occurs.

Notification Callback API

The Notification Callback will be used to inform the different SmartSociety platform components of the messages that arrived for them (e.g., to inform the Task Execution Engine about task results or other task-related information (e.g., an error)).

As multiple SmartSociety components may expose this API, the Middleware will try to determine the exact recipient components by reading the message fields and checking the routing rules. However, in some cases (e.g., asynchronous input messages from peers) the Middleware may not be able to determine the exact recipient components, and will forward the message to all the SmartSociety platform components implementing the API. There, a requirement for those components is that they be capable of handling (filtering) unexpected messages.

Method	**Description**
notify	Notifies the Task Execution Engine about task results or task-relation information like an error.

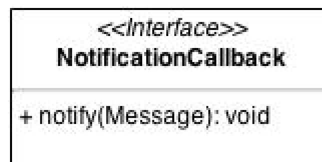

Figure 18: Notification Callback interface

notify
Notifies the Task Execution Engine about task results or task-relation information like an erro Return none Parameters <table><tr><th>Parameter</th><th>Type</th><th>Description</th></tr><tr><td>message</td><td>Message</td><td>the received message. As explained, the Message may contain the Execution ID or other information that allows the Middleware to route the message.</td></tr></table> Throws nothing

Collective Info API

This API is used to provide different information regarding the composition and the state of the collectives to the Middleware, in order for the Middleware to allow to other SmartSociety components the functionality of addressing their messages on the Collective level.

At this point, the API consists of a single method, but as the TEE and EPE components get developed later, the API may grow and/or change.

Method	Description
resolveCollective	Resolves and returns the members of a given collective.

Figure 19: Communication Callback

resolveCollective

Resolves and returns the members of a given collective id.

Return
List of peer ids that are part of the collective.

Parameters

Parameter	Type	Description
collectiveId	String	The id of the collective

Throws
NoSuchCollectiveException - if there exists no such collective.

Internal Functioning and Interaction with SmartSociety Platform Components

In the following we will discuss a scenario to describe the interaction with and within the system.

Motivating scenario

Let us consider a municipality which wants to leverage SmartSociety platform (SSP) to build and deploy a distributed application for emergency response. A SmartSociety application Smart Response (SR) is developed and deployed on SSP's Task Execution Engine . Citizens wanting to participate are encouraged to register, and either download a dedicated client app (SRPeer-App) on their devices, or participate (even concurrently) via web interface (SRPeer-Web), either from mobile as well as from stationary devices. By registering, the citizens (i.e. peers from now on) become eligible to participate and to be invited to join different possible collectives belonging to SR application.

We will first describe a general scenario to help better understand the usage context. The full scenario is provided in the already circulated D7.1 draft. We will then consider a subset of it as a particular use-case that will showcase the Middleware's functioning and the interaction with the SSP components.

General scenario

A local flooding occurred. Part of the municipal territory is covered with water. Part of the road connections are therefore cut off. Some citizens may have stayed or could not move out of their buildings/houses. Transportation is affected. Certain objects may be endangered - buildings, passages, bridges. Citizen collectives are dispatched to track and maintain a real-time map of viable ground routes and take pictures that help assess the state of the important objects.

We can distinguish between two types of communication in this scenario:

a) **Collective lifecycle management communication** –The communication needed to manage collective formation, task negotiations, monitoring, gathering (sub-)results, rewarding. This communication is happening mostly between the SSP and the peers. SSP wants to be in full control over the exchange of such messages, and thus controls how and when such messages are exchanged.

b) **Peer coordination and collaboration communication** – The freely exchanged messages among the peers working on an execution of a particular task. The SSP is not interested in controlling these messages, but in order to facilitate the collaboration is interested in setting up the communication channels. This communication is mostly peer-to-peer through third party services.

While collective's lifecycle management communication is managed through SmartSociety's components, the peer-to-peer collaboration is preferably executed with the support of well-known external tools. Therefore, each collective is provided with an external virtual collaboration and coordination environment. For example, a collective is given an url to access a Google map on which they can enter markers (road under water, house damaged, people inside), credentials for a cloud-based storage service (e.g., Dropbox/ownCloud) where they can upload the GPS-annotated pictures of the objects, as well as a url of a chat room/twitter hashtag/Hangout where they can freely communicate between each other.

Use-case

Peers need to be notified that their assistance is needed. We will assume that no task negotiations are needed, as the peers decide to help of a voluntary base. The peers get notified through the channel they marked as preferred in their peer profiles; they have a choice of SMS or email for that. Peers are given a list of locations that they should visit and photograph. Upon successful photographing, the pictures need to be uploaded to a given Dropbox folder. If the peer cannot reach a given object (e.g., due to flooding or rubble), he is asked to inform the SSP by sending an email to a dedicated mailing list with a predefined subject line containing the name of the unreachable object.

The Task Execution Engine (TEE) asks to its Elastic Provisioning Engine (EPE) that a collective be composed for the use-case execution. The EPE assembles and returns a collective with an unique ID. The EPE keeps managing the collective's lifecycle. The Middleware is responsible for handling the communication with the collective.

In order to initially notify the peers about participating in the task execution, the TEE sends a message addressed to the collective using the Communication API which serves as the entry point for the Middleware. The Middleware resolves the current collective members through the Collective Info API, and for each of the peers reads the preferred communication method by contacting the Peer Manager (PM) through PM Callback API. Some peers want to be notified by SMS, the others by email. Let us consider first those that prefer to be contacted through SMS. The Middleware reads the phone number of a peer, and decides through which mobile operator to send the SMS. Those peers that prefer to be contacted through email will be sent an email. The contents of the message are provided by the TEE. In this case, the message contains the streets that need to be checked, as well as the Dropbox URL and Dropbox access tokens. The SMS's or emails are delivered to the peers by the respective Output Adapters, described later in more detail. Output adapters form part of the Middleware, but as they serve as connectors for different communication technologies and protocols, they sometimes may need to be provisioned from the TEE through Communication API. This will be the case for some non-standard adapters that will not come originally as part of the Middleware. We can assume

that some common output adapters will be provided by the middleware itself (e.g., SMTP email output adapter, SRPeer-App adapter, SRPeer-Web adapter).

For the sake of simplicity, we assume that peers do accept to go and perform the required task. Once on the given location, the peer takes the picture and at some point uploads it to the given Dropbox url authenticating with the access token. A Input Adapter (part of the Middleware) regularly checks the Dropbox folder. When a new picture is discovered, it informs the TEE of it, by sending the response to the TEE associated with the Execution ID waiting on it via notify method in the Notification Callback API. The Input Adapter was registered with the Middleware by the TEE at the very beginning of the task execution through the Communication API. Similarly to the Output Adapters, some Input adapters will come prepackaged with the Middleware (e.g., email POP, Dropbox, Twitter) while if TEE will require additional context-specific functionality, it would have to provide the implementation and provision the input adapter prior to task execution.

So far, we have seen how the input information from the peers arrives back to the TEE. In the case of Dropbox polling the peers do not actively send the information. On the other hand, if a peer needs to inform the TEE that he cannot reach an object to take picture of it because the object is inaccessible, the peer will want to actively send a message to the SmartResponse application executing on the SSP. In this case, the use-case foresees that the peer send a message with a predefined subject line to a dedicated email address. He does so using his favorite email application. A Input Adapter that listens to that mail server inspects the received message, and decides whether to forward it to the TEE, converting it to the internal message format, thus effectively performing message filtering and transformation.

We will now show how to perform certain common functionalities using the presented APIs in more detail, referring to the presented use case as the running example where needed.

Creation of output adapters

In order to create a output adapter that can send task requests to the peers, the adapter has to implement the OutputAdapter interface. For the purpose of demonstration the following description we will focus on a communication using SMS (the implementation of the email adapter would be similar to the SMS adapter), any other form of communication (e.g., a dedicated mobile application, email, instant messaging) could be used. The SMS adapter has to contact a SMS provider to send messages to the mobile phone number of peers.

Note that the name of the adapter has to be unique in order to be able to resolve the address of peers and the preferred communication adapter. Stateful adapters will be created per peer, meaning that every peer is associated with a separate adapter instance. Both parameters have to be added using the @Adapter annotation, which is required for every OutputAdapter.

```
@Adapter(name="SMS", stateful = false)
public class SMSAdapter implements OutputAdapter {

   public SMSAdapter() {
      //initialize the adapter for a (multiple) given provider
   }

   @Override
   public void push(Message message, PeerAddress address) {
      //send message to address
   }
}
```

Listing 1: Implementation of a Output Adapter

To tell the Middleware to use the adapter, its type has to be registered in the Communication interface to be available for peers.

```
//register the output adapter
communicationInstance.registerOutputAdapter(SMSAdapter.class);
```

Listing 2: Registration of a Output Adapter

Creation of input adapters

In the previous chapter we discussed two different types of input adapters that are available. The first one should regularly check if there are new pictures in the Dropbox folder, therefore we need a input pull adapter for Dropbox. The other one will receive emails from the mailing list by push.

Note that the both adapters have to be provided with the concrete parameters (i.e., Dropbox folder and credentials as well as address of the mailing list and credentials). This information can either be hardcoded or provided at the creation of the adapter in the Task Execution Engine or at a higher level.

```
private class DropboxAdapter implements InputPullAdapter {

  private DropboxAdapter(String pullAddress, String credentials) {
      //initialize adapter
   }

   @Override
   public Message pull() {
      //perform pull and transform to message
```

```
        return msg;
    }
}
```

Listing 3: Implementation of a Input Pull Adapter

```
private class MailinglistAdapter extends InputPushAdapterImpl {
    Thread thread;

   private MailinglistAdapter(String pullAddress, String credentials) {
        //initialize adapter
    }

    @Override
    public void init() {
        thread = new Thread() {
           public void run() {
               //wait for message and transform to internal message
               publishMessage(msg);
           }
        };
        thread.start();
    }

    @Override
    public void preDestroy() {
        thread.interrupt();
        //destroy other stuff
    }
}
```

Listing 4: Implementation of two Input Adapter

Both adapters have to be instantiated and added using the Communication interface.

```
//create and add the input adapters
communicationInstance.addPushAdapter(new MailinglistAdapter(address));
communicationInstance.addPullAdapter(new DropboxAdapter(anotherAddress));
```

Listing 5: Instantiation of input adapters

Send messages to peers and collectives

After the adapters have been created and registered (i.e., added) using the Communication API, the Middleware can be used to send messages to peers and collectives. Therefore a message associated with a execution id and a specific type and a receiver (can be a specific peer or a collective) has to be created. Further information, like content, time-to-live and so on can be specified but are not mandatory for the interaction with the Middleware.

A provided MessageFactory supports the creation of messages and prohibits illegal usage.

```
String executionId = //specify id
String type = //specify type
String receiver = //specify receiver (peer or collective)

//create message
Message msg = MessageFactory.create(executionId, type, receiver);

//add further message details

//send message to peers
communicationInstance.send(msg);
```

Listing 6: Sending a message to peers

Receive input

The component that should receive input (the Task Execution Engine in our case) has to implement the Notification Callback API in order to be notified of newly arrived input. As soon as the Middleware receives input from any of its input adapters it decides how to route the message and who should receive it. If the receiver is the Task Execution Engine, its notify method will be called.