FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Virtualizing Communication for Hybrid and Diversity-Aware Collective Adaptive Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Philipp Zeppezauer, BSc
Matrikelnummer 0926320

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dr.techn. Hong-Linh Truong

Wien, 02.12.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Virtualizing Communication for Hybrid and Diversity-Aware Collective Adaptive Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Philipp Zeppezauer, BSc
Registration Number 0926320

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dr.techn. Hong-Linh Truong

Vienna, 02.12.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Philipp Zeppezauer, BSc
Herrnaugasse 16, 5020 Salzburg

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I wish to thank various people for their contribution to this Master's thesis. Hong-Linh Truong, my supervisor, for his guidance, enthusiastic encouragement and useful critiques of this thesis, and everyone who contributed to the Smart Society Project. I am particularly grateful for the guidance and assistance given by Dipl.-Ing. Ognjen Scekic throughout the creation of the Master's thesis and the project. I would also like to express my thanks to the various members of the Distributed Systems Group at the University of Technology Vienna.

Finally, I wish to thank my family and especially my parents for their support and encouragement throughout my study.

This work is partially supported by the EU FP7 SmartSociety project under grant № 600854.

Abstract

Hybrid and Diversity-Aware Collective Adaptive Systems (HDA-CAS) form a broad class of highly distributed systems comprising a number of heterogeneous human-based and machine-based computing (service) units. These units are required to perform tasks on their own and in cooperation with other units to solve complex problems. Therefore, they typically interact and collaborate in an ad-hoc manner. These units form dynamic adaptive collectives, which are subject to constant change. Whenever possible, the collectives are allowed to self-orchestrate, using familiar collaboration tools and environments.

The flexibility of these collectives makes them suitable for processing elaborate tasks. At the same time, building a system to support diverse types of communication types in such collectives is challenging, because the way how human-based and machine-based units communicate differs fundamentally. To be able to use both in a hybrid system, the actual way of communication between the units has to be virtualized and handled in the system independently of the actual type of the communication participants allowing for a uniform communication between applications of the HDA-CAS platform and individual service units.

In this thesis, the fundamental communication challenges for HDA-CAS are addressed and requirements, and properties of communication in a HDA-CAS are formulated. This thesis discusses these problems and presents a concept of how to virtualize the communication with service units and collectives. Therefore, the notion of service units is extended and the concept of communication adapters is discussed. Furthermore, this thesis presents a design of a middleware which uses the concepts for virtualizing the communication between, within and among collectives and service units. The middleware is able to handle numerous, intermittently available, human and machine-based service units, and manage the notion of collectivity transparently to the programmer. A prototype implementation for validation and evaluation purposes is also provided.

Kurzfassung

Hybrid and Diversity-Aware Collective Adaptive Systems (HDA-CAS) bilden eine umfassende Klasse von hochgradig verteilten Systemen, die aus einer Vielzahl von heterogenen Computing (Service) Units, basierend auf Menschen oder Maschinen, bestehen. Diese Service Units erledigen Aufgaben sowohl selbständig, als auch in Kooperation mit anderen Service Units um komplexe Probleme zu lösen. Daher interagieren und arbeiten sie in der Regel in einer Ad-hoc-Weise zusammen. Diese Service Units bilden dynamische, adaptive Kollektive (so genannte 'Collectives'), die einem ständigen Wandel unterworfen sind. Wann immer möglich, ist es den Collectives erlaubt, sich selbst zu orchestrieren und vertraute Tools und Umgebungen für die Zusammenarbeit zu nutzen.

Durch ihre Flexibilität sind diese Collectives für die Verarbeitung aufwendige Aufgaben geeignet. Jedoch ist der Aufbau eines Systems, das verschiedenen Arten von Kommunikationstypen in einem solchen Collective erlaubt, eine Herausforderung, weil sich die Art und Weise, wie menschliche und maschinelle Service Units kommunizieren, grundlegend unterscheidet. Um beide Arten in einem hybriden System nutzen zu können, muss die Kommunikation mit diesen virtualisiert werden und unabhängig des tatsächlichen Typs behandelt werden. Dies erlaubt eine einheitliche Kommunikation zwischen der HDA-CAS Plattform und den individuellen Service Units.

In dieser Diplomarbeit werden die grundlegenden Herausforderungen der Kommunikation in HDA-CAS besprochen und Anforderungen und Eigenschaften der Kommunikation in einem HDA-CAS formuliert. Diese Arbeit diskutiert diese Probleme und stellt ein Konzept vor, wie die Kommunikation mit den Service Units und Collectives virtualisiert werden kann. Daher wird der Begriff der Service Units erweitert und das Konzept der Communication Adapters wird diskutiert. Darüber hinaus stellt diese Arbeit die Architektur und das Design einer Middleware vor, die die Konzepte für die Virtualisierung der Kommunikation zwischen und innerhalb der Collectives und Service Units verwendet. Die Middleware ist in der Lage, zahlreiche menschliche und maschinelle Service Units und Collective transparent für den Programmierer zu behandeln. Eine Prototyp-Implementierung für die Validierung und Bewertung des Designs wird ebenfalls vorgestellt.

Contents

CHAPTER 1

Introduction

Collective Adaptive System (CAS) [20] is a term for highly distributed systems consisting of numerous autonomous computing elements, each with individual properties and preferences, but supporting the fundamental property of collectiveness. *Collectiveness* implies that the individual elements need to communicate and collaborate in order to reach common decisions, or perform tasks jointly. To provide collectiveness, CASs gather sets of multiple computing elements into so called *collectives*, that provide additional functionality compared to single computing elements due to their collective capabilities [32]. *Adaptiveness* is another basic property of CASs, defining that at any given time computing elements are allowed to join and leave the system, and collective compositions as well as task execution goals can be dynamically altered. This implies that CAS are open systems, subject to constant change.

CASs come in a variety of forms; [32] defines that in research there are ”bio-inspired and self-organizing branches, evolutionary and adaptive-control strategies, different software and hardware approaches”. This thesis focuses on one specific form of CAS: Hybrid and Diversity-Aware Collective Adaptive Systems. **Hybrid and Diversity-Aware Collective Adaptive Systems (HDA-CASs)** [25] additionally add the *heterogeneity* to the founding principles of CASs. This means that they inherently support communication and collaboration among different types of computing elements, such as software/machines, people and things (e.g., sensors). To support heterogeneity, the platform has to virtualize the communication with computing elements and communicate regardless of whether they are humans, machines or things, as well as regardless of the application that makes use of such computing elements.

To support the development of this field the European Union funded the collaborative project SmartSociety[1] [38], comprised of ten universities and institutes. The goal of this project is to build a HDA-CAS that combines and virtualizes humans, machines and things to build a smarter society.

This thesis will discuss the problems of communication that emerge when building a Hybrid and Diversity-Aware Collective Adaptive System that incorporates humans, machines, and

[1]Full title: ”Hybrid and Diversity-Aware Collective Adaptive Systems: When People Meet Machines to Build a Smarter Society“. `http://www.smart-society-project.eu/`

things (e.g., sensors, actuators). In the main part of the thesis, a system – called SmartCom – will be presented and discussed in detail which claims to solve these problems. In the following this system will be referred to as 'Communication Middleware' or just 'Middleware'. Although the presented middleware is intended to be used within the SmartSociety platform, it is not specifically designed for this single purpose and is therefore generally applicable as a communication middleware to a wide number of similar HDA-CAS platforms. Therefore, this thesis will only make explicit references to the SmartSociety platform within the introduction of the thesis and will simply use the term 'platform' afterwards to emphasize the general applicability. In addition, components of the HDA-CAS platform will be called 'platform components'.

1.1 Problem Statement

Figure 1.1 shows the high-level architecture of the SmartSociety platform and presents the middleware's operational context. The SmartSociety platform supports the programming and execution of computations involving humans, machines and things. The *users* of the platform (e.g., a smart-city maintenance provider; see Section 1.2 for details) submit complex tasks to an application running on the platform – the so called *SmartSociety application*. The application performs the task – with the help of features of the platform, e.g., orchestration – by assembling and engaging **collectives** of **service units** to execute the (sub-)tasks collaboratively. A *service unit* [43] is an entity that consists of a peer (human, machine, or things) and a context (the concept of service units will be described in detail in Section 3.2.1).

The way how human-based and machine-based units (e.g., software) communicate differs fundamentally. While inter-machine communication relies on well-defined concepts, technologies, and protocols on multiple layers (e.g., TCP, or REST), human communication in a digital environment is unconstrained and supported by different communication tools (e.g., email, or Social Networks). To be able to use both in a hybrid system, the actual way of communication between the units has to be virtualized and handled in the system independently of the actual type of the communication participants.

1.2 Motivating Scenarios

The following section takes a look at two motivating scenarios. The first one deals with disaster/crisis management [13, 17], such as a flooding, and how an HDA-CAS can support the management thereof by supporting communication among heterogeneous types of service units, such as flooding experts, medical workers and common citizens. The second part of this section presents a scenario that deals with a smart-city and the maintenance work thereof which also requires a lot of communication between equipment manufacturers, coordination and monitoring centers, and maintenance workers.

Further interesting scenarios in the context of CAS would be, for example, a tourist guidance application driven by locals to recommend events and sights, a restaurant/bar recommendation application driven by locals, or a car-sharing application. [2]

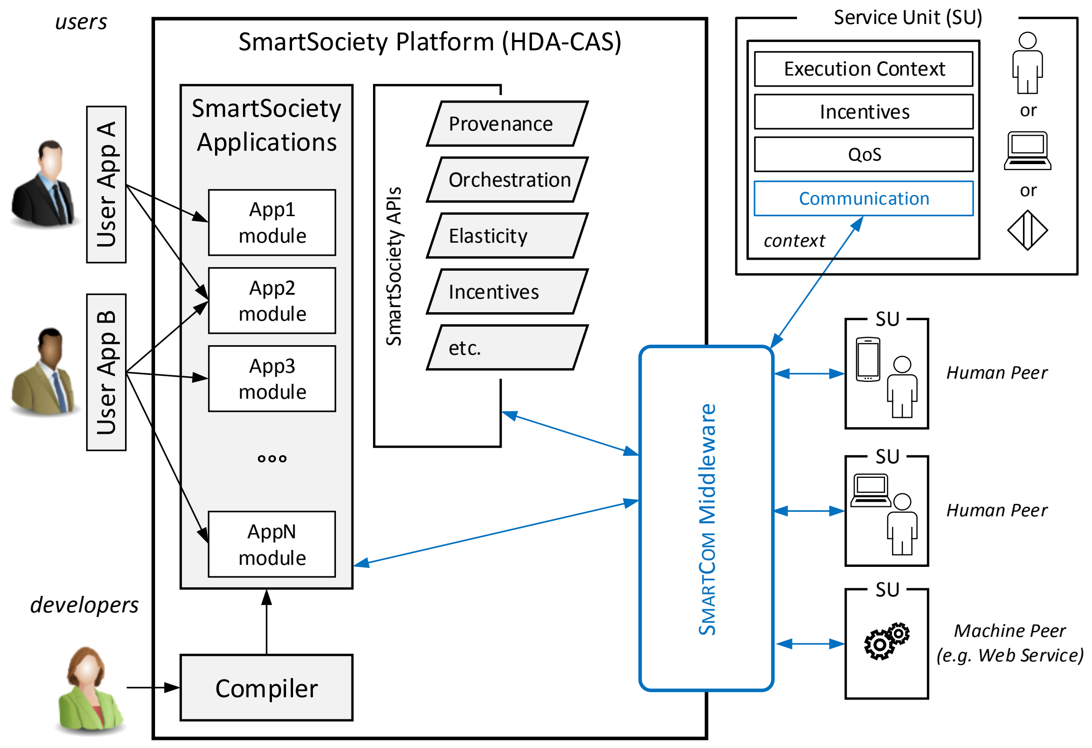


Figure 1.1: Operational context for the middleware. Components belonging to the communication middleware are marked in blue.

1.2.1 Disaster Management

Let us consider a municipality which wants to leverage a HDA-CAS platform to build and deploy a distributed application for emergency response (e.g., a massive flooding in a city and surrounding areas). A dedicated application is developed and deployed on top of the platform on the HDA-CAS platform. Interested citizens can participate by registering their phone number via a web interface to receive SMS notifications, or download a specific peer application on their smartphones. Further possibilities would be to register their email address or any other form of communication. By registering, the citizens become human-based service units and are eligible to participate and to be invited to join different possible collectives belonging to the emergency response application. The system also provides the possibility to register as an expert in a certain field (e.g., boat owner, medical worker, radio amateur). The goal of the application is to utilize the capabilities and the mobility of these service units to enhance the emergency response management. Besides humans also software units can participate in the application and provide certain functionality, like Complex Event Processing or Data Analysis on data provided by weather stations and sensor observing the water level in rivers to detect dangerous situations.

Now let us consider that the weather data indicates heavy rain and the water level of the river flowing through the municipality will be critical within hours. This situation is detected by the data analysis component which immediately starts the emergency procedure. Collectives of ordinary citizens are formed and asked to evaluate the situation in their neighborhood. All of these service units are notified using their preferred method (e.g., SMS, or email) or on their dedicated application. In case of an increasing water level, the system automatically forms further collectives which are asked to reinforce the existing dams with additional sand bags.

However, despite the effort in reinforcing the dams, the water levels keep rising and parts of the municipality are flooded. As the status of road connections and citizens trapped in their homes is unclear, further assistance from citizens is required. Therefore, all registered citizens are combined to local collectives and are requested to update a Google Map[2] and provide information about the status of road connections and trapped citizens. Additionally they can upload a picture to a Dropbox[3] folder with a geo-tag. Besides the regular communication channels, a Twitter[4] post is created that encourages unregistered citizens to participate and respond to the request using a specific hashtag. This data (i.e., updated statuses on the map, pictures in the Dropbox folder and responses on Twitter) is automatically collected by the platform and provided to the authorities to rescue citizens and improve the handling of the disaster.

This scenario has been chosen because it is already well-known and incorporates all important aspects and functionalities that are required for the virtualization of communication on such a platform: sending messages to service units and collectives, selecting the communication channels for individual service units, interactions between humans and machines, and collecting feedback data from various communication channel.

1.2.2 Smart-City Maintenance

Let us consider a smart-city that consists of numerous geographically dispersed buildings and infrastructure facilities each with thousands of sensors that monitor these buildings and facilities. And let us consider a smart-city *maintenance provider* (*MP*), a company running a monitoring center that collects and analyzes data of these sensors (e.g., Pacific Controls Galaxy[5]).

The MP provides the centralized service of both *predictive* and *corrective* maintenance to its customers (building/equipment owners/tenants). This means that MP control centers actively monitor events originating from various sensors and perform Complex Event Processing on these data flows. If a potential or actual malfunction is detected they dispatch collectives of experts to analyze the situation in detail. If necessary, the physical maintenance work is performed on the ground by a collective of technicians. The (human) experts are contracted to work on-demand with the MP, subject to their availability and the actual work performed by them. These collectives consist of multiple human units as well as software units that support their work, each of these units with own preferences on communication.

Since each equipment manufacturer has different issue analysis and reparation procedures, when equipment from different manufacturers is interconnected in a smart building, detecting

[2]https://www.google.at/maps
[3]https://www.dropbox.com/
[4]https://twitter.com/
[5]http://www.pacific-galaxy.com/

the cause of an anomaly event sequence cannot easily be done by following prescribed procedures. The complexity grows further when considered at the scale of a smart city, with thousands of building, each with a unique equipment mix, age, environment, and agreed service-level. Therefore, a conventional workflow type of orchestration does not fit well for such a scenario. Rather, collectives of human experts perform loosely-controlled collaboration patterns in order to detect and repair the problem in the most efficient way, considering the particular context, and making use of supporting software tools when needed (e.g., for data analysis, communication).

1.3 Results

The goal of this thesis is the theoretical principles for supporting virtualization and communication within described Hybrid and Diversity-Aware Collective Adaptive System (HDA-CAS) platforms. Furthermore, the thesis presents requirements, as well as an architecture, design, application programming interfaces (APIs) and a prototype implementation of a middleware solving the problems. The proposed system provides the communication and virtualization primitives to support heterogeneity, collectivity and adaptiveness.

In addition the system provides native support for virtualizing collectives by hiding the complexity of communication with a dynamic collective as a whole and passing of instructions from the HDA-CAS execution engine to it, making it a first-class, programmable entity. This native support allows to communicate with the collective members transparently, regardless of whether they are human-based or machine-based (e.g., sensors, or software). Single human, sensor or software services are able to participate in different collectives concurrently, acting as different service units with different SLA, delivery and privacy policies.

The proposed middleware supports collaboration patterns on a higher level. A collaboration pattern controls the effort within a collective in a loose manner. Instead of over-regulating interactions, the collaboration patterns set the limits within which the service units are allowed to self-organize, using familiar collaboration tools and environments.

A collaboration pattern consists of the following elements:

- *Relationship topology* – specifying different topologies (e.g., independent, tree, ring, sink, random) and relation types formalizing relationships among service units in a collective. The meaning of the relations is application specific, and can be used to express communication, data, or command flow.
- *Collaboration environment* – specifying access to familiar external tools that the service units can use to collaborate among themselves (e.g., Google Docs, Dropbox). When a collective is formed, service units are provided with instructions and appropriate access credentials for the previously set up collaboration environment.
- *Communication channels* – analogously to the collaboration environment, the pattern should specify access to familiar external tools that the service units can use to communicate among themselves and with SmartSociety Platform.
- *Security and privacy policies* – policies to restrict the communication and interaction with specific (sub)collectives or with predefined communication channels.

- *Delivery policies* – policies to control how messages are delivered to service units using their communication channels and preferences.

In addition the system will provide services that can be used by the HDA-CAS platform to enforce incentives [34] or to track provenance [9] (i.e., tracking the origins and the processing of data) within the communication middleware.

The results of this thesis are partially published at the 10th International Workshop on Engineering Service-Oriented Applications (WESOA'14) in Paris, France in November 2014 [47].

1.4 Structure of the Thesis

This thesis is broken into six chapters. The introduction discusses the problem and the motivation of the thesis and introduces some common terms of the research field of collective adaptive systems. Following the introduction, Chapter 2 discusses the state of the art in communication in related research fields. Furthermore, we will take a look at common open-source enterprise service busses because their functionalities are related to these that are required by the problem statement.

Chapter 3 discusses the key concepts for virtualization of communication in HDA-CAS. Chapter 4 presents the design of a middleware as the suggested solution to the problem stated in the introduction of this thesis. First, the architecture of the proposed system is examined and the individual components are described in detail regarding their purpose and functionality. Furthermore, the mapping of the stated problem to the suggested solution is discussed. Additional service components that support the communication but do not provide core functionality are described at the end of the first section. Following the architecture of the system, the communication using messages is described in Section 4.2. This section discusses the structure of messages, how they are routed and the predefined messages used in the system. Section 4.3 discusses the application programming interfaces of the components described in the beginning of the chapter. The following section (Section 4.4) describes some algorithms that are used by components of the systems and finally the last section of this chapter (Section 5.1) discusses briefly the implementation of the prototype of the middleware.

In Chapter 5, the suggested solution is evaluated by means of one of the motivating scenarios, and some design decision, functional and non-functional requirements, are discussed. Furthermore, some small performance evaluations are presented. Finally Chapter 6 concludes this thesis and provides an outlook on future work.

CHAPTER 2

State of the Art

The research field of HDA-CAS is new, but highly varied, and in its many niches builds upon different founding technologies and research results, spanning research areas, such as: autonomous agents in multi-agent based systems, robotics, bio-inspired collectives, human-provided services, crowdsourcing and web-scale workflow technologies. This thesis only focuses on the communication aspect, in particular on the communication with hybrid units (i.e., humans, machines and things). To the best of the knowledge of the author, no other platforms or middleware systems offer the virtualization for communication of collectives, humans and machines in a HDA-CAS in a similar manner. This section discusses the state of the art in handling the communication and virtualization between entities (e.g., service units and a system) in multiple HDA-CAS-related research fields. The end of this chapter discusses some popular enterprise service busses that are used for the communication and the interaction of multiple components using different communication styles. Existing approaches are reviewed and compared to the requirements defined in Chapter 1 and the proposed middleware presented in Chapter 4.

2.1 Multi-agent Systems

Multi-agent Systems (MAS) are systems that consist of multiple computing elements (called agents) that interact with each other [44]. Agents are characterized by the following two capabilities [44]: *i*) autonomous actions (i.e., deciding what they have to do themselves to reach their objectives); and *ii*) the interaction with other agents to cooperate, coordinate, negotiate and exchange data.

One of the crucial communication aspects of multi-agent based systems is the agent communication language that is used by the peers to understand each other. The papers [14, 33] give an overview of the landscape of agent communication languages (ACLs) and discusses two fully-specified languages: FIPA ACL and KQML. KQML [21] defines three layers: a content layer, a message layer and a communication layer. The content layer contains application specific content in any format. The message layer defines how the message are handled (e.g., if it is a negotiation, or a query) and further details like the ontology, or the language of the message. Finally the communication layer defines how the communication should be handled by defining the receiver, sender and further parameters. FIPA ACL [22] has been developed by the “Foundation for Intelligent Physical Agents” (FIPA), an ”IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies” [3]. The FIPA ACL specification has been used for example in the Java Agent DEvelopment (JADE) Framework [7]. The FIPA ACL defines a set of message types which can be used for the interaction of agents, but message contents are application specific. According to [33] both languages are almost identical.

In [10], authors propose a middleware that supports communication among agents on different platforms and programming languages. They use a different runtime for each platform and exchange messages between these runtimes to achieve a cross-platform communication of agents.

GAIA [40] is a distributed middleware infrastructure for active spaces (i.e., physical environments with lots of user interaction with a large variety of devices, i.e., an office). It acts as a meta-operating system supporting the development and execution of portable applications. It coordinates software units as well as heterogeneous network devices. The middleware can also be used to register and query services within an active space. Internally it uses CORBA [27] for the interaction between individual units but it is also possible to provide a customized implementation for the communication between units.

The Context Toolkit [19] is a context-aware middleware. It supports the development and deployment of context-aware applications. A system using the Context Toolkit consists of context widges (i.e., software components providing access to context information) and a distributed infrastructure that hosts these widgets. Communication between the widges is handled by web-standards such as HTTP and XML.

The PACE middleware [28] is a middleware for complex, heterogeneous, and context-aware distributed systems. The whole middleware consists of a context management system, a preference management system, and a programming toolkit. The internal messaging framework allows the communication between application components and middleware services. Interface definitions are mapped to stubs which handle the communication (e.g., stubs for RPC). These stubs can be generated for various programming languages and technologies which makes the system very flexible. Note that all nodes in the system have to use the same stub to be able to communicate with each other.

Compared to the approach in this thesis, MAS communication focuses only on peer-to-peer interactions, lacking interactions with a managing system that needs to impose specific communication patterns or privacy constraints. Therefore, MAS systems fully rely on the semantics built into the language for this. Additionally, agents in MAS are usually uniform, and the agent type is known in advance, while a middleware for a HDA-CAS platform has to support service units using different communication channels during runtime.

Similarly to middlewares of multi-agent system, which offer automated transformation between agent languages, the middleware also needs to offer message transformation between different message formats used in HDA-CAS. The logical message model in the middleware addresses the same abstraction layers as the mentioned ACLs – the application specific contents of the message is encapsulated into a message format that dictates different delivery and privacy policies, while personal preferences of peers and availability of communication channels dictate how the actual message delivery is performed at the communication layer. The content layer is managed by the sender, the message layer by the middleware, while the communication layer is managed also by the middleware but on behalf of the receiver.

2.2 Swarm Robotics

Swarm Robotics is inspired by social insects like bees and aims at the coordination of a large number of simple robots [4]. The work on peer communication is multifaceted, usually in a homogeneous environment (all robots are of the same kind) and only peer-to-peer. In the ASCENS project [45] (focusing on collective adaptive system) one of the communication strategies relies on the visual communication. [11] describes such a visual communication where many homogeneous robots have a common task and try to solve it collectively. The robots have to be able to see each other and use different lights to indicate the task they are working on. Conflicts on task selection are handled between the conflicting peers. The coordination is decentralised and handled in a peer-to-peer manner.

Besides this approach, there are also some swarm robotic projects/algorithms which completely avoid explicit communication among peers, for example in [31].

Since this research area is inspired by nature, the agents tend to use physical signals to perform communication, and self-organize, exhibiting collective intelligence. This area served mostly as inspiration to make the ”natural” human communication tools a first-class citizens of proposed middleware presented in Chapter 4, and to support unmanaged peer-to-peer communication. In practice this means to support communication and collaboration tools that human peers are likely to use in their every-day lives and integrate them with the middleware in hope that this attracts humans to use HDA-CAS platforms more easily and more naturally, but also to exploit a wide array of existing functionalities these tools offer. For example, instead of having to install a dedicated application to receive a request, the user can simply use its existing mail/Twitter application to receive the request, and upload the result of his/her work on Dropbox to share it with others. In this way, users need not learn to use additional applications, and can also leverage the infrastructure resources and implementational maturity of existing tools.

2.3 Service-based Systems

The ALLOW Ensembles project deals with the concept of cell ensembles [6]. These ensembles consist of cells that are given a declaratively-defined behaviour, but the actual workflows, that they execute, adapts during runtime depending on the collective (ensemble) goal. The focus here is on adaptation of workflows to achieve the adaptability of CAS, whereas the virtualization and communication with human and machine elements is performed through standardized Web Service (WS–*) technologies. They use BPEL4Chor [18] for inter-cellular choreography. Using WS-BPEL4People [42] or Adaptive Pervasive Flows [8] one can also incorporate human activities.

The ASCENS project focuses on the peer-to-peer approach in machine-only ensembles/collectives (e.g., robots, vehicles, storage nodes) [46]. Similarly to SmartSociety, the ASCENS models the fundamental constructs as service-based components, however, due to the specificity of coordination languages they use, at the communication layer they use Pastry [41] and extend it with the SCRIBE protocol [12] to support message routing and delivery in a peer-to-peer fashion via any- and multicasts [36]. This behavior differs from the approach of this thesis since we provide a centralized middleware for message exchange, while relying on an adaptive concept of centrally managed collective.

2.4 Crowdsourcing Platforms

Crowdsourcing Platforms like Jabberwocky [5], TurKit [35] or CrowdLang [37] utilize human capabilities to solve problems.

Jabberwockys computing stack consists of Dormouse, ManReduce and Dog. Dormouse provides the functionality to interact with humans and machines using a platform-independent programming environment that sits on top of other crowdsourcing platforms. ManReduce follows the MapReduce paradigm but enables the programmer to decide whether to use a human or machine in both, the map and the reduce phase. Finally Dog is a high-level scripting language that makes use of ManReduce and has been created to increase the flexibility of the programming environment. [5]

Amazon provides a web service called Amazon Mechanical Turk [1] that allows to issue small human tasks, so called Human Intelligent Tasks (HITs). Human workers can solve these tasks and get a monetary reward after finishing successfully. Usually these tasks are independent of each other and can be created and solved in parallel. Turkit provides a programming environment on top of Amazon's Mechanical Turk that enables the programmer to define a workflow of tasks. It is able to collect the results of finished tasks and create further tasks based on these results. [35]

However, in reality they are frameworks/components/libraries layered on top of existing human-computation commercial platforms, such as Amazon Mechanical Turk, Clickworker, CrowdFlower. This implies that fundamentally, the virtualization and communication on the lowest level is limited by the functionalities offered by the underlying platform. The focus is put on (adapting) the workflow to be executed. For each workflow action the underlying platform's API is used to offer the corresponding human task to the crowd. In this respect, to a platform user, the virtualized concept of the peer in these systems is a programming language construct describing peer's capabilities, constraints, and promised rewards which are passed on to the underlying platform which ultimately provisions the peers to perform the task. The concept of collectives is not supported, neither are unconstrained peer-to-peer communication.

2.5 Enterprise Service Busses (ESBs)

An Enterprise Service Bus (ESB) [15] is a software architecture that aims to handle the communication between various software applications and components of a system (e.g., in an enterprise) in a service-oriented architecture. Another important purpose of an ESB is the integration of several systems. The following section discusses a few popular Enterprise Service Busses because they provide similar functionality as the presented middleware in Chapter 4. The following discussion of popular open-source ESBs is based on the comparison in [16] in 2011 with respect to the currently available versions. Due to the requirements for virtualization of communication in an HDA-CAS, we will only take a look at messaging, integration with adapters, privacy and delivery policies, access control, and multi-tenancy of Mule ESB (3.5.0) [39], Apache ServiceMix (5.1.1) [24], JBoss ESB (4.12) [30], and WSO2 ESB (4.8.1) [29].

MuleESB provides patterns for message routing but does not provide message normalization within the ESB, messages are translated only as needed. As many other ESBs MuleESB does also support custom adapters and is shipped with many predefined adapters. It fully supports different security protocols for access control. The ESB does also provide some kind of multi-tenancy but does not enforce any policies at runtime. [16, 39]

Apache ServiceMix provides various patterns for message routing and uses normalized messages to integrate components. By using Apache Camel[1] various adapters can be used by Apache ServiceMix to integrate components. Furthermore, it is possible to define custom adapters to adopt the ESB to new technology. Apache ServiceMix does not support any privacy or delivery policies that can be applied to specific components and it seems that it does not provide multi-tenancy as well. The ESB provides different security methods using JAAS to provide access control to the system. [16, 24]

JBoss ESB also provides various patterns for message routing and uses normalized messages too. It also supports the creation of custom adapters but they are not as flexible due to some restrictions compared to other ESB providers. JBoss ESB provides access control for different components and also a (limited) support for policies. The ESB does also provide some kind of multi-tenancy. [16, 30]

WSO2 ESB does support all functionality mentioned above except the dynamically enforcement of policies and is - considering only the discussed requirements - the most advanced ESB available. [16, 29]

In general all discussed middlewares provide adapters to communicate with components and custom adapters to adopt to new technology advancement. All ESBs, except MulESB, use normalized messages internally for the messaging and routing. In general none of the presented ESBs comes with support of addressing of collectives which is one of the key features of the proposed middleware in this thesis. Furthermore, the support of humans interacting with the system is generally not considered.

[1] `http://camel.apache.org/`

CHAPTER 3

Virtualizing Communication

This chapter presents the requirements as well as the key concepts for the virtualization of communication in Hybrid and Diversity-Aware Collective Adaptive Systems (HDA-CASs).

3.1 Requirements for Communication

A middleware supporting communication between humans, machines, and things in a HDA-CAS has to fulfill the following requirements, which have been gathered by examining the properties of HDA-CAS and possible interactions of peers and components of such a system:

1. **Virtualization** – supporting heterogeneous human-based and machine-based service units as uniformly addressable entities; Supporting ‘collective’ as a first-class, dynamically-defined entity.
2. **Heterogeneity** – supporting various types of communication channels (protocols) between the platform and service units as well as among service units/collectives, transparently to the platform.
3. **Communication** – providing primitives for: message transformation, routing, delivery with configurable options (e.g., retry, expiry, delay, or acknowledgment). Allowing to send unicast messages to single service units as well as multicast messages to multiple service units that are part of a collective.
4. **Scalability** – ability to handle large number of intermittently available service units.
5. **Extensibility** – ability to extend the system with further communication channels introduced due to technological advancements in the future.
6. **Persistence** – message persistence and querying of messages to analyze and derive metrics/incentives [34], or to gain further insights in the handling of communication.
7. **Asynchronous Communication** – due to the high response time of humans (i.e., minutes to hours or even days compared to milliseconds/seconds of machine-based service units)

8. **Security** – even if not a key functionality, it should be possible to provide primitive authentication of messages and service units, as well as a simple session management.

The communication with service units has to be handled independently of their actual type, allowing for a uniform communication between applications of the platform and individual service units. Platform components should be able to send messages to service units which are identified by a unique identifier. The mapping of the identifier to the communication channel of service units should be handled by the middleware with the assistance of platform components that are managing service units and collectives. Since service units are not limited to a single communication channel, there might be multiple mappings of identifiers of service units to communication channels (e.g., a service unit can be contacted using a mobile application and/or SMS, or access the system via a mobile application and a web browser concurrently).

3.2 Key Concepts

The following sections present the key concepts of virtualizing communication in HDA-CAS. The first section extends the notion of a service unit, which describes a peer and a context. The second section takes a look at communication adapters and how they are essential to fulfill the expected requirements.

3.2.1 Service Unit

The notion of a *service unit* is based on the concepts introduced in [43]. A service unit is an entity provisioned and utilized through service models (e.g., on-demand and pay-per-use). As presented in Figure 3.1, a service unit consists of:

- *Peer* – abstraction of a physical or virtual entity performing the computation or executing a task. Can be a human, a machine, or a thing (e.g., a sensor). If not explicitly mentioned, the notion of a peer always describes the whole service unit in the following.
- *Context* – a set of parameters describing the execution context of the particular HDA-CAS platform and the applications in which the service unit is participating.

The context parameters can include: execution Id, QoS requirements, performance metrics, associated incentives [34]. To describe the communication with a service unit, the context also consists of a *Communication Context*. The communication context defines the context of conversations, and negotiations in which the service units participate, as well as context-dependent virtualization and communication channels (e.g., using email, SMS). This context is important for communication since each service unit has its own preferences on communication, e.g., some humans prefer email, others a dedicated mobile application.

In general service units can use different communication channels to interact with the HDA-CAS platform, e.g., a human-based service unit can communicate via email and Twitter interchangeably, receive task descriptions and track progress through a web application, and communicate with other units within the collective through a dedicated mobile application. Alternatively service units can make use of external software services, serving as collaborative and

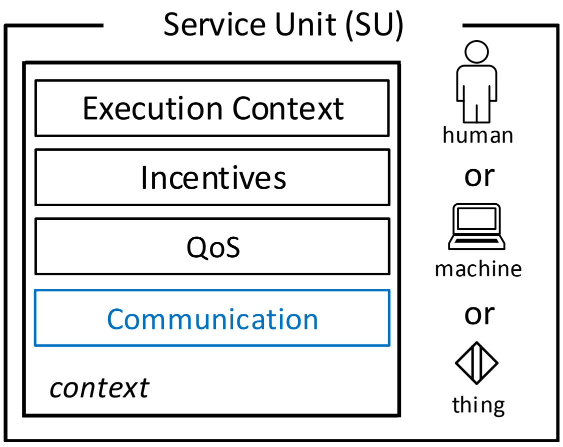


Figure 3.1: Concept of a service unit. It consists of a peer and a context.

utility tools. For example, a software service like Doodle[1] can be used to agree upon participation times, or Dropbox as a common repository for performed tasks. Both human-based and machine-based service units can drive the task processing, e.g., a software may invoke workflow activities which are performed by human-based service units; or, conversely, human-based service units can orchestrate the execution independently, using software services as data analytics, collaboration and coordination tools.

A *collective* is a dynamic entity that consists of multiple service units in order to reach common decision or to perform tasks jointly. At any given time, service units are allowed to join or leave collectives, which makes them highly adaptive and dynamic.

3.2.2 Communication Adapter

The concept of *Communication Adapters* (simply called 'adapters') is important for the proposed middleware since it is essential to fulfill the expected requirements. This section describes the adapters from the functional perspective. Section 4.1.1 takes a look at the technical perspective of adapters.

Functionally the adapters are components that handle the communication with a service unit (peer in the following) over a certain communication channel either by sending a message (e.g., by sending an email to the peer's email address) or by receiving feedback from peers (e.g., a peer responded to a mailing list). The middleware uses adapters to abstract and virtualize peers to the rest of the platform. The concept of an adapter allows for:

1. **Hybridity** – by enabling different communication channels to and from peers. Adapters effectively handle the communication and provide the abstraction of the peer's concrete type and communication preferences.

[1]`http://doodle.com/`

2. **Scalability** – by enabling the middleware to cater to the dynamically changing number of peers and workload during the execution.

3. **Extensibility** – new types of communication and collaboration channels can easily be added at a later stage transparently to the rest of the HDA-CAS platform.

4. **Usability** – human-based service units are not forced to use dedicated applications for collaboration, but rather freely communicate and (self-)organize among themselves by relying on familiar third-party tools. Adapters can provide access to or utilize such tools in order to handle the communication with peers.

5. **Load Reduction and Resilience** – by requiring that all the feedback from peers goes exclusively and unidirectionally through external tools first, only to be channeled/filtered later through a dedicated input adapter, the middleware is effectively shielded from unwanted traffic load, delegating the initial traffic impact to the infrastructure of the external tools. At the same time, failure of a single adapter does not affect the overall functioning and availability of the middleware.

CHAPTER 4

Communication Middleware Design

This chapter presents a communication middleware called *SmartCom*, which uses the concepts of the previous chapter to provide virtualization of communication to a Hybrid and Diversity-Aware Collective Adaptive System (HDA-CAS) platform. In the following this system will be refer to as *middleware*. The primary goal of the middleware is to virtualize and handle the communication between platform components of the HDA-CAS, applications and peers, and among peers themselves. In this chapter, the notion of a peer always describes the whole service unit as described in Section 3.2.1.

Section 4.1 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 4.2 examines how the middleware handles messages between the platform, the applications, the middleware, collectives, and the peers, and how messages look like. Section 4.3 presents the application programming interfaces (APIs) of the components that have been described in Section 4.1. Finally, Section 4.4 outlines some more complex algorithms that are used by the system.

4.1 Architecture

This section presents the various components of the middleware that are responsible for the virtualization and the handling of communication with peers in a HDA-CAS. Each component is described in detail regarding its purpose, its functionalities and how they interact with other components of the system. The implementation of the proposed system and components is discussed briefly in Chapter 5.

Figure 4.1 presents a conceptual overview of the proposed middleware. Platform components on the left hand side usually initialize the communication and messages are sent to peers and collectives on the right hand side by utilizing the middleware for the communication. Such outgoing messages are called *Output Messages*. Peers can reply to received messages by sending messages themselves to the middleware. Additionally they can use additional tools (e.g.,

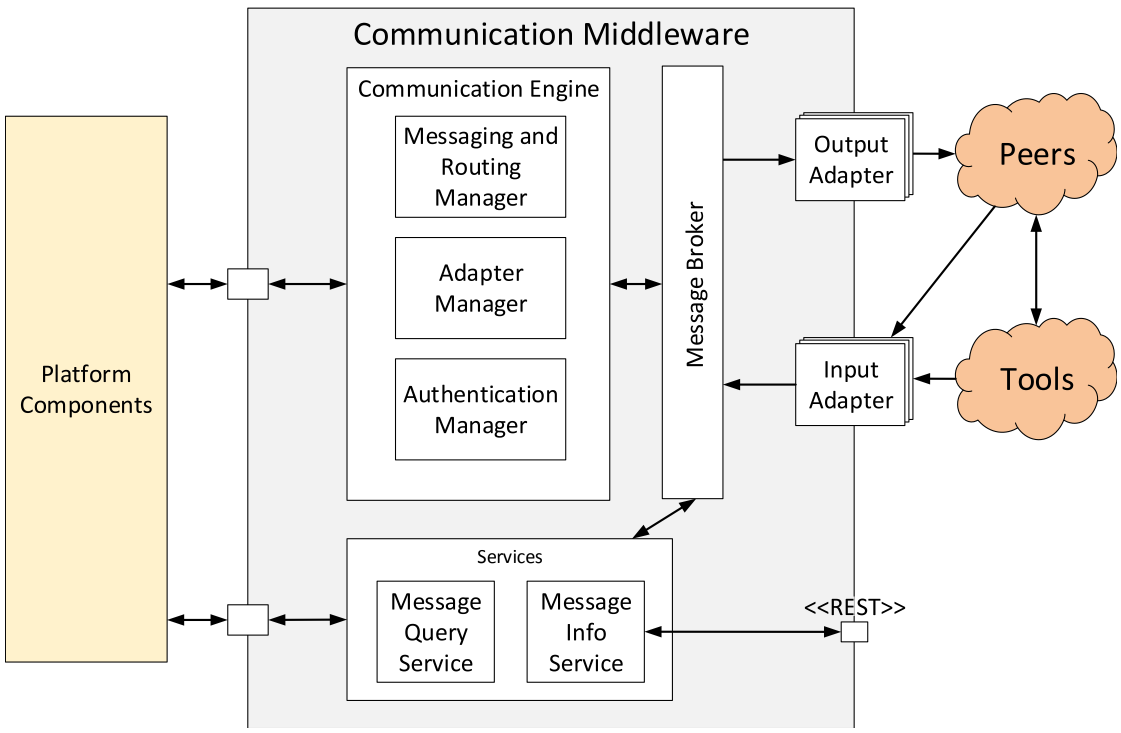


Figure 4.1: Overview of the communication middleware. Platform Components are part of the HDA-CAS platform and peers/tools are external to the system.

uploading a file to a file server) which are monitored by the middleware (i.e., the system checks regular if there are updates/changes). These ingoing messages are called *Input Messages*. A communication between different peers is also supported, but they are not required to communicate with each other solely by using the middleware. This approach allows the wide spread integration of workflows and HDA-CAS applications with the middleware because it does not limit already existing communication patterns among peers (e.g., a telephone call) by enforcing own models but rather supports additional communication and interaction if required.

Being a first-class citizen, interactions with, within and among collectives are also supported. However, this communication is handled on the peer-level for each peer that is part of the collective. Due to this reason, collectives are not mentioned explicitly in most components described below. Collectives are only handled specifically at the initialization of a communication in the Messaging and Routing Manager (see Section 4.1.2). At this point the members of a collective are resolved and handlers are registered to detect successful and unsuccessful communication attempts.

The component in the middle of the Figure 4.1 represents the middleware and its various subcomponents. The box on the left hand side labeled as 'Platform components' indicates external components, which are part of the HDA-CAS platform, as well as applications running on a HDA-CAS. Peers, and tools used by peers (e.g., a file server or a mailing list) are also considered as being external to the system.

The middleware consists of four groups of components:

- **Adapters** are used to handle and abstract the actual communication with peers and tools over communication channels (e.g., sending an email, SMS or making a REST call) from the rest of the middleware 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 which are intended to be sent to or received from peers and tools. These components are responsible to resolve the members of collectives and to initialize the communication. Additional functionalities like message authentication, the management and execution of adapters, and routing of messages are also provided by the components of the Communication Engine.

- **Services** support the peers, platform components of the HDA-CAS and applications, and provide additional information on communication aspects. Furthermore, they can be used – for example – to derive metrics (e.g., average response time) and profiles [26] for peers or to supervise/monitor the communication.

- **Message Broker** is used to decouple the execution of the various components of the system and to handle bigger workloads by supporting 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 fully detailed diagram of all the components working together is presented in Figure 4.2.

4.1.1 Adapters

The functional aspect of adapters has been described in Section 3.2.2. This section discusses the technical aspect of adapters within the middleware. 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 contact a peer (i.e., send a message to a peer) and the Input Adapter is only allowed to receive input messages from an external tool or peer.

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.

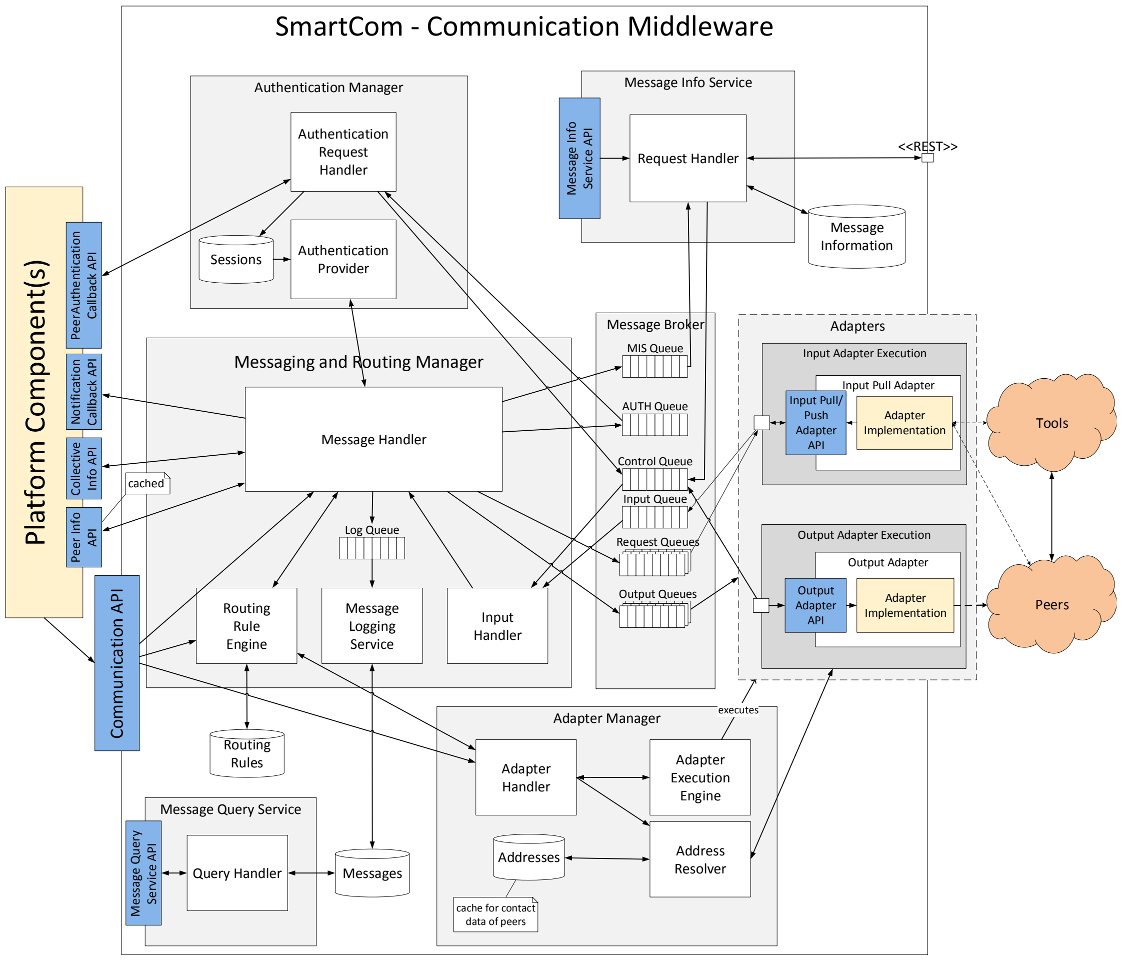


Figure 4.2: Detailed view of the communication middleware.

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 4.1.2). On the other hand, Input Adapters are created by applications running on the platform and are passed to the middleware 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.

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

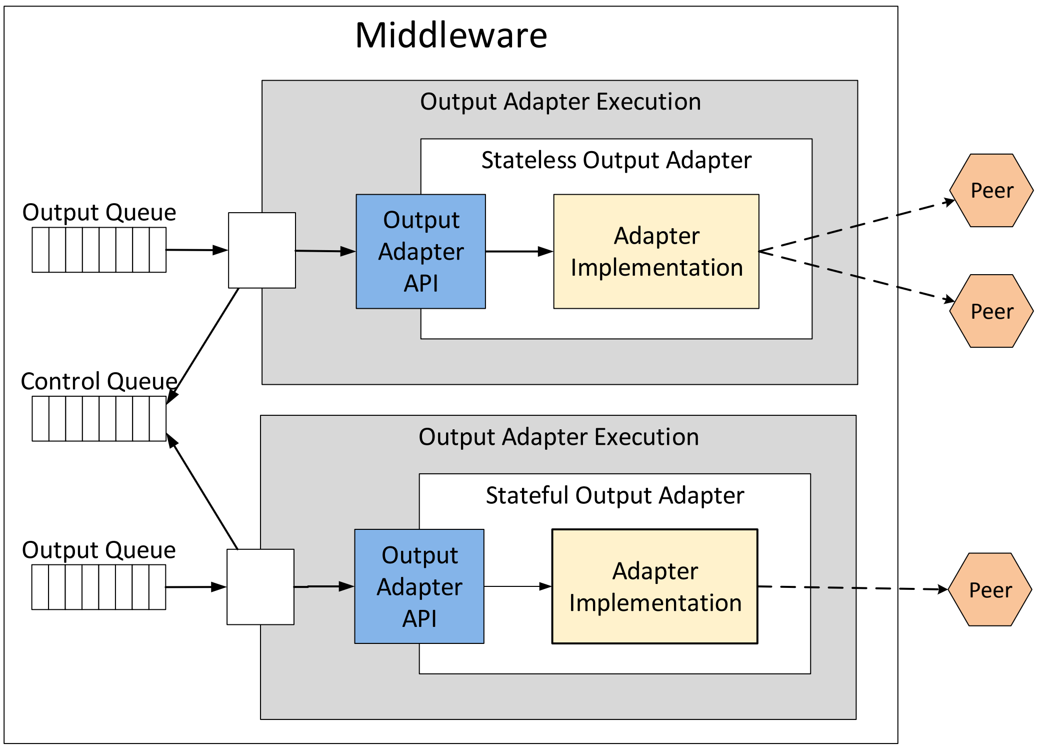


Figure 4.3: Concept of the Output Adapter

Output 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 4.3 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 the middleware that the sending was successful or of an error. Instances of both categories have to implement the `Output Adapter API` (see Section 4.3.2).

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 4.1.2).

- 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 the middleware 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.

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 4.4 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 4.3.2), instances of Input Pull Adapters have to implement the `Input Pull Adapter API` (see also Section 4.3.2).

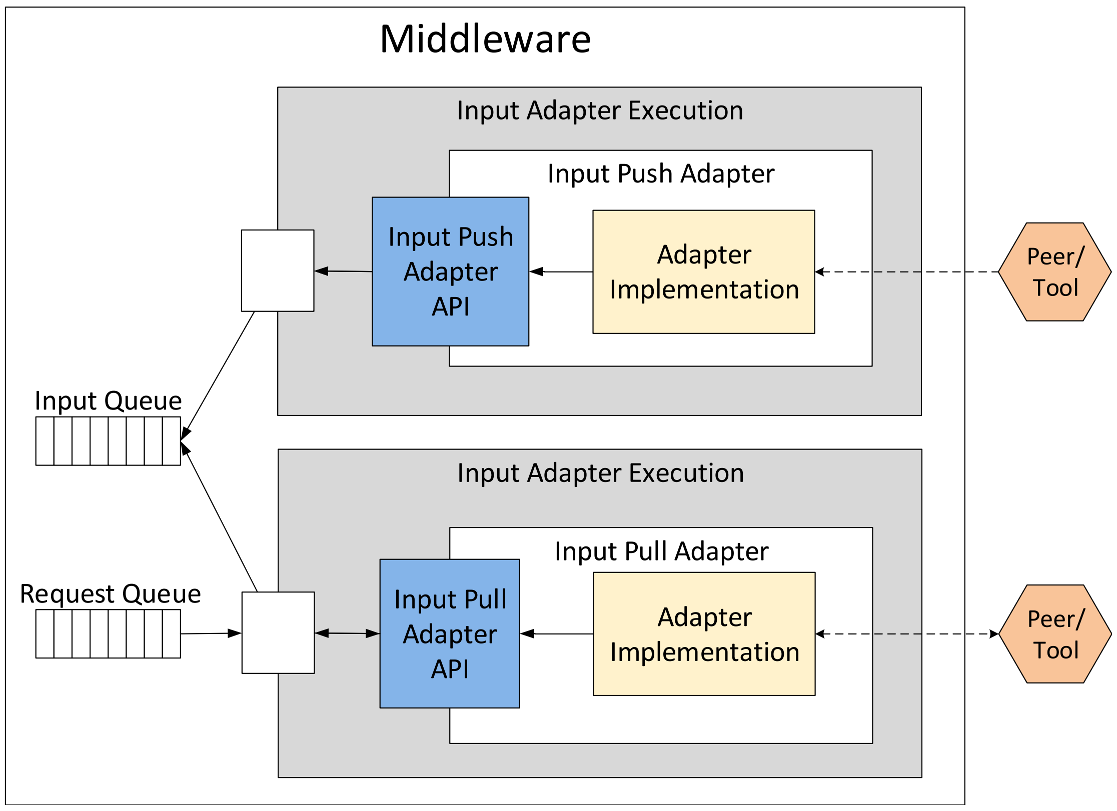


Figure 4.4: Concept of Input Push Adapters and Input Pull Adapters.

The lifecycle of Input Adapters completely depends on applications. They are created and removed by applications because their configuration is application specific. Consider an Input Adapter that checks a FTP server for new files and each application requires an application-specific directory to be observed. This configuration cannot be managed by the middleware because it highly depends on applications and the concrete technology used. Therefore, the instantiation and removal of Input Adapters is handled by applications and not by the middleware.

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.

4.1.2 Communication Engine

The Communication Engine is the core of the execution system of the middleware 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, Authen-

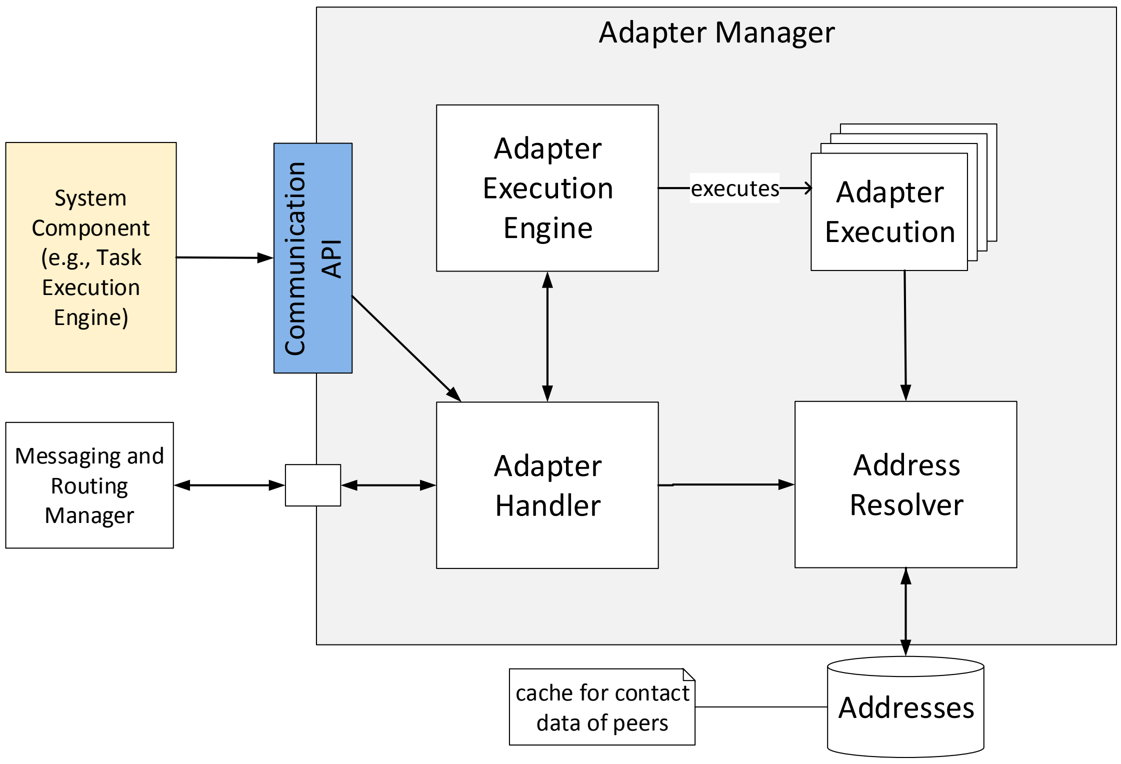


Figure 4.5: Subcomponents of the Adapter Manager and interaction with other internal and external components.

tication Manager, and the Messaging and Routing Manager. The interactions of the subcomponents can be examined in Figure 4.2.

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 peers using corresponding Input and Output Adapters (described in Section 4.1.1), 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 the middleware to verify the sender of a message. These components are described in the following sections.

Adapter Manager

The Adapter Manager is responsible for the lifecycle management (i.e., registration, creation, initialization, execution, and removal) of adapters intended for the communication with peers. Adapters are responsible for sending and receiving messages from peers using a communication channel (e.g., SMS or email). Further details on adapters can be found in Section 4.1.1 and Section 3.2.2. The Adapter Manager consists of the following subcomponents which are described below: Adapter Execution Engine, Adapter Handler, Address Resolver and multiple Adapter Executions. Figure 4.5 shows the internal structure of the Adapter Manager and how the subcomponents interact with each other.

The **Adapter Handler** manages the registered but not yet instantiated instances of Output Adapters. Both categories of Output Adapters are registered at the Adapter Handler, although Stateless Output Adapters are instantiated immediately, Stateful Output Adapters are just 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 an 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*[1]. 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 but rather executed immediately and handled according to their category (i.e., push or pull adapter). The detailed algorithm can be examined in Section 4.4.1.

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 4.3) 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.

To support scaling out to handle big workloads, each Output Adapter initially listens for new messages on a single, adapter specific Output Queue of the Message Broker (see Section 4.1.3). 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 as 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 Stateful Output Adapters are per 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.

Platform components and applications have to create instances of Input Adapters themselves and pass these instances to the middleware using the Communication API (see Section 4.3). 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

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

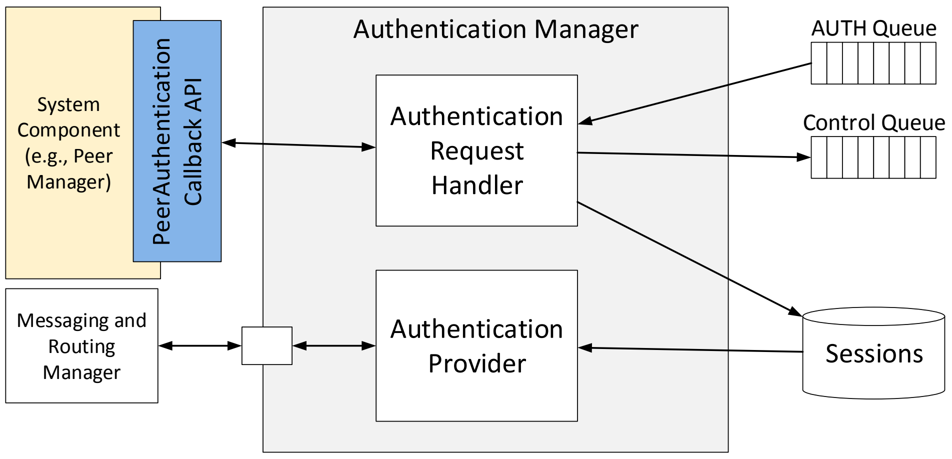

System Component (e.g., Peer Manager)
PeerAuthentication Callback API
Messaging and Routing Manager
Authentication Manager
Authentication Request Handler
Authentication Provider
AUTH Queue
Control Queue
Sessions

Figure 4.6: Concept of the Authentication Manager

message represented by a new resource (e.g., a new file) available. On the other hand, platform components are not able to create instances of Output Adapters directly. Such instances are created on demand because - unlike Input Adapters - they are shared among all applications running on a HDA-CAS platform.

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 factoring regarding performance and throughput.

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 4.2.3) 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 4.3.3) 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 the middleware 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

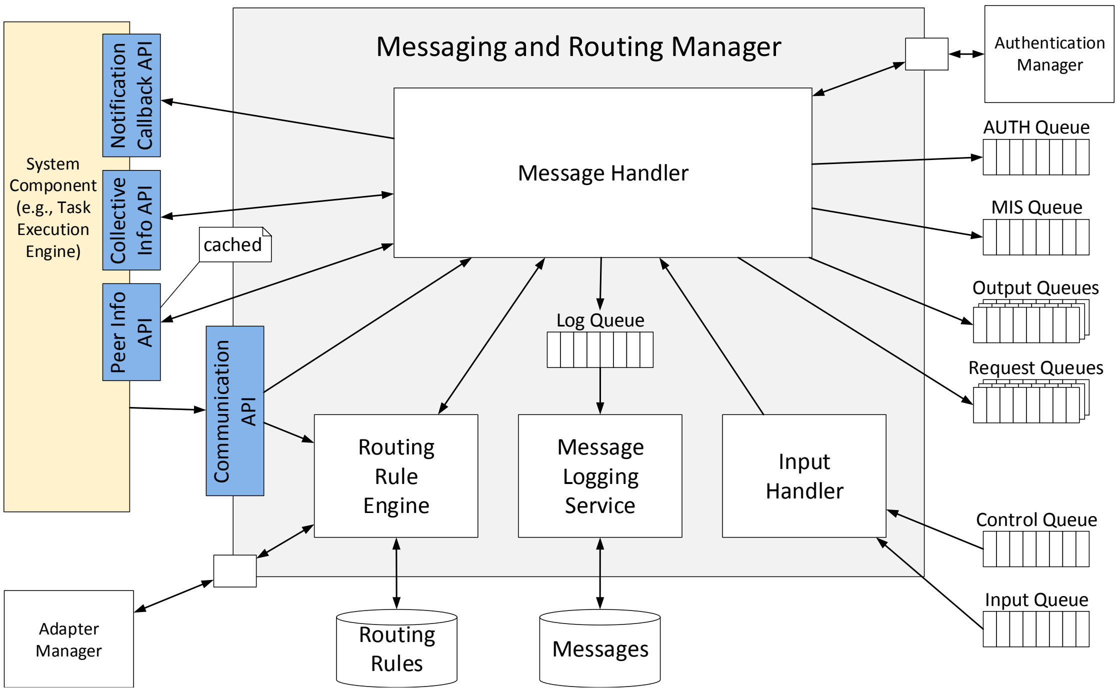


Figure 4.7: Concept of the Messaging and Routing Manager

verify the authenticity of a message – if required. Figure 4.6 presents the internal structure of the Authentication Manager.

The Authentication Manager uses a **Sessions 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. This timestamp is used to invalidate a session after a predefined period of time. Whether this time stamp is updated upon each usage of the session or just at the beginning of the session is up to the implementation of the Authentication Manager. 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.

Messaging and Routing Manager

The Messaging and Routing Manager is responsible for the handling of internal and external messages of the system. Figure 4.7 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 4.3.3) to retrieve the information about a peer which contains the communication channels used by the peer. Since this information does not change very often, it can be cached to improve the performance of further requests. In the following, the Adapter Manager is instructed to create new adapters according to the peer's 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 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 4.3.3). 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 4.3.3) 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 the middleware.

The complete algorithm can be found in Section 4.4.2. Platform components can register themselves in 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 the middleware.

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. Further details on routing are discussed in the following section about routing.

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 the logging of all sent and received messages to a persistent database. This information can be used to debug the middleware, or to analyze the data and create – for example – incentives or profiles of peers. This information can be retrieved by using the Message Query Service (see Section 4.1.4).

Routing There are two types of routing available in the proposed middleware. 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 is 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 null together. This restriction prohibits that all messages of the system are forwarded to a single peer. Besides that there are no further restrictions on routing rules.

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. Note that routing rules also impose a potential security risk to applications, it allows – for example – to easily implement eavesdropping on private conversation. The presented routing is also not multi-tenancy aware which also imposes a security risk because routing rules from an application A might also match messages from an application B. Future versions of the proposed middleware should address this problem in particular.

Handling of Policies

The middleware 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 the middleware and are retrieved by calling the `Peer Info API` (see Section 4.3.3).

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 the middleware.

2. The second level of delivery policies is concerned with the peer delivery policies. Besides specifying the peer's communication channels in the peer's profile (in an external component), it can also specify how it has 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 available. The enforcement of this 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 acknowledgment 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 send the message, whereas TO_ANY requires only 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 and highest 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 4.3.3) 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.

4.1.3 Message Broker

The purpose of the Message Broker is to decouple the executions of the various components of the middleware. Furthermore, it is used to implement the first type of routing (see Section 4.1.2) 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**: is 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.

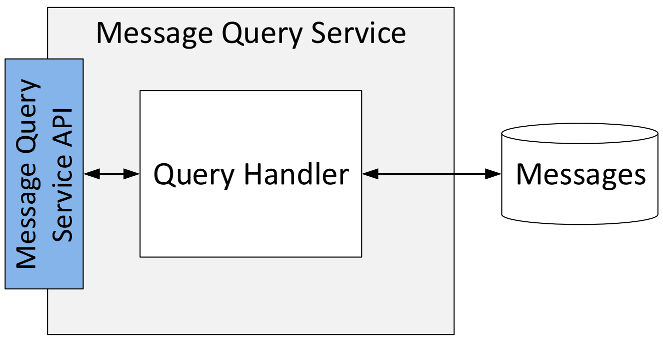


Figure 4.8: Concept of the Message Query Service

- **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**: are 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**: is a special queue for messages that are intended for the Authentication Manager. Messages in this queue are authentication request messages (see Section 4.2.3) which consist of the username and password so that peers can be authenticated.
- **MIS Queue**: is a special queue for messages that are intended for the Message Info Service. These messages are usually request messages (see Section 4.2.3) for information about a certain message indicated by a certain type and subtype.
- **Log Queue**: is intended for all messages that are handled within the middleware 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.

4.1.4 Services

Message Query Service

The Message Query Service provides an interface to query sent and received messages. Figure 4.8 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 the middleware.

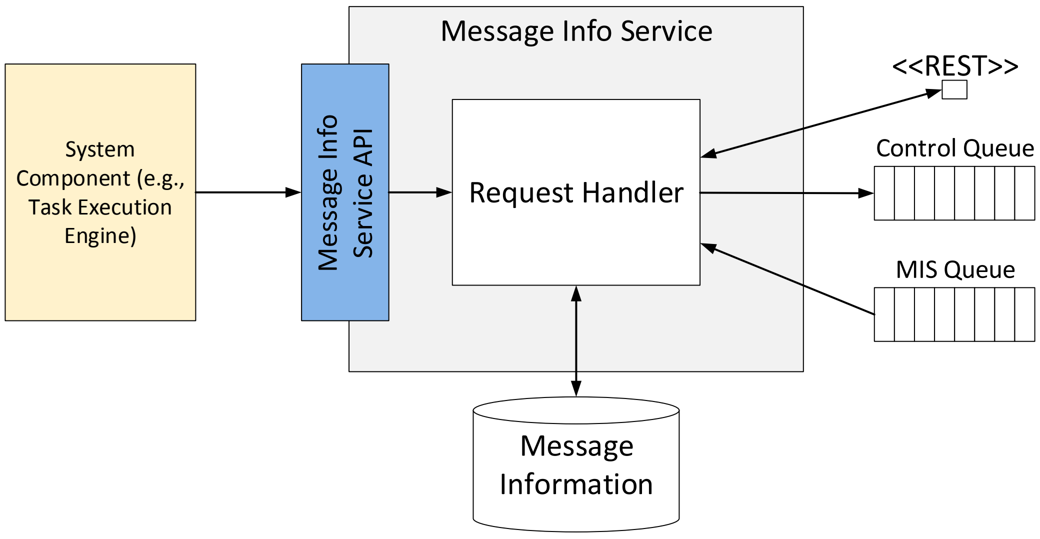


Figure 4.9: Concept of the Message Info Service

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 (described in Section 5.1) provides 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 is updated through platform components. Figure 4.9 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 4.2.3) 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 4.3.2).

4.2 Messages

The middleware exchanges messages with platform components, the adapters as well as with some internal components (i.e., the Authentication Manager and the Message Info Service). Therefore, the notion of messages is important. 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.

4.2.1 Message Structure

The following section presents the structure of messages that are used within the middleware. The structure is quite similar to the FIPA ACL Message Structure [23], but some properties have been removed and others added to fit the requirements of the middleware.

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 4.1. Listing 4.1 outlines a simple message containing instructions for a task in the JSON format.

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

Listing 4.1: 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 the middleware does not use the conversation-Id internally, this functionality has to be provided by a platform component.

4.2.2 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 4.1.2.

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 a 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 the middleware 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 4.1: Structure of messages. Optional fields are marked with (o).

4.2.3 Predefined Messages

The middleware defines some 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.

Control Messages

Control messages are exchanged within the middleware 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., acknowledgment of a message) or exceptions during the communication. Their are described in detail below. Table 4.2 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 doesn't 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 4.2: Predefined subtypes of Control Messages.

Message Info Messages

These messages are handled by the Message Info Service (see Section 4.1.4). 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 4.3 presents the two subtypes of message info messages.

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).

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 4.3: Message Info Request and Reply Messages.

Such messages are handled by the Authentication Manager (see Section 4.1.2) which interacts with a platform component to verify the identify of a peer. Further information can be found in section 4.1.2. 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 the middleware to the peer. Table 4.4 describes the used subtypes.

Message	**Description**
Authentication Request (*REQUEST*)	Authentication request message of a peer that contains its credentials. The Authentication Manager queries a 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 the middleware 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 the middleware 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 the middleware 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 4.4: Authentication Messages.

4.3 Application Programming Interfaces (APIs)

The following section takes a look at the API of the middleware. 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 the middleware 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.

4.3.1 Data Structures

Table 4.5 presents the data structures that are exchanged between the middleware and the platform components. They are mainly used by the public entities described in Section 4.3.2, and the callback entities described in Section 4.3.3.

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, the middleware and peers. There are also internal messages that are just handled between middleware components, or applications and middleware components. See Section 4.2 for details.
RoutingRule	Defines a rule of how messages should be handled within the middleware. 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 4.2.2 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 4.1.1 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 4.5: Domain model and data structures of the middleware.

4.3.2 Public Entities

Table 4.6 describes the interfaces that are exposed by the middleware 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.

Entity	**Description**
Communication	Main entity that is used for the communication with the middleware. 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 4.6: Public entities of the middleware that are used to interact with the system.

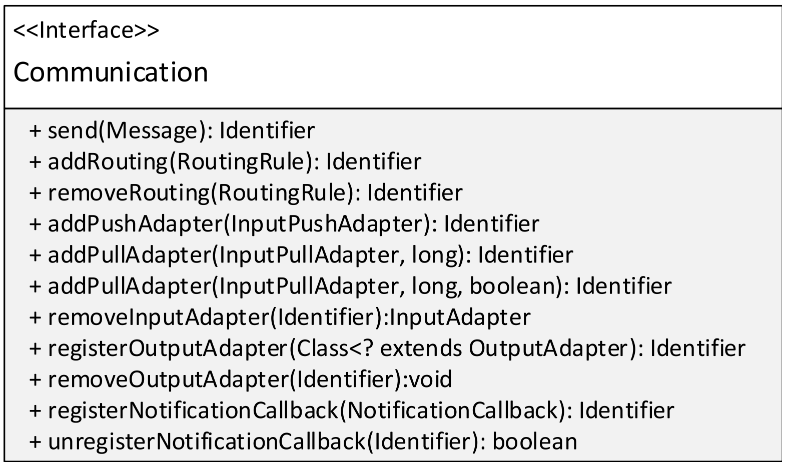
Communication API

This section discusses the main API for the interaction with peers, collectives, and the middleware 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 the middleware. Figure 4.10 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 4.3.3). 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 Notification Callback API immediately.

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


<<Interface>>

Communication

```
+ send(Message): Identifier
+ addRouting(RoutingRule): Identifier
+ removeRouting(RoutingRule): Identifier
+ addPushAdapter(InputPushAdapter): Identifier
+ addPullAdapter(InputPullAdapter, long): Identifier
+ addPullAdapter(InputPullAdapter, long, boolean): Identifier
+ removeInputAdapter(Identifier):InputAdapter
+ registerOutputAdapter(Class<? extends OutputAdapter): Identifier
+ removeOutputAdapter(Identifier):void
+ registerNotificationCallback(NotificationCallback): Identifier
+ unregisterNotificationCallback(Identifier): boolean
```

Figure 4.10: Communication API

Returns

Returns the internal Id of the middleware 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). The middleware 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 the middleware.

Returns

Returns the middleware 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 the middleware 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 the middleware 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 deleteIf-Successful 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 the middleware 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 the middleware to get in contact with a peer. The output adapters are instantiated by the middleware 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 the middleware 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 removed 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 the middleware 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 implemented. Output Adapters receive a message from the middleware, 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 4.1.1 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 the middleware. Figure 4.11 presents the Output Adapter API in UML notation.

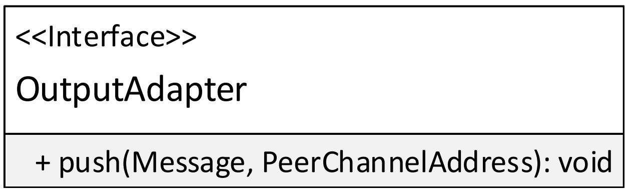


Figure 4.11: Output Adapter API

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).

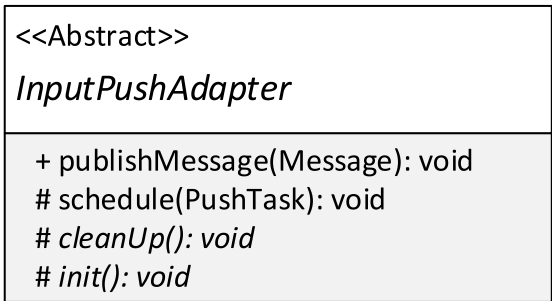


Figure 4.12: Input Push Adapter API

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 the middleware 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 4.1.2), 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

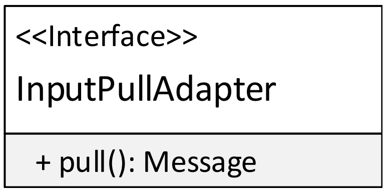


Figure 4.13: Input Pull Adapter API

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.

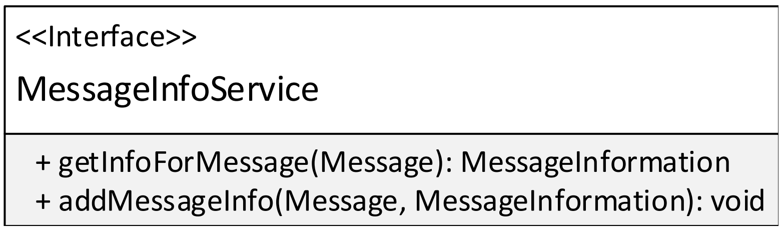

<<Interface>>
MessageInfoService
+ getInfoForMessage(Message): MessageInformation
+ addMessageInfo(Message, MessageInformation): void

Figure 4.14: 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.

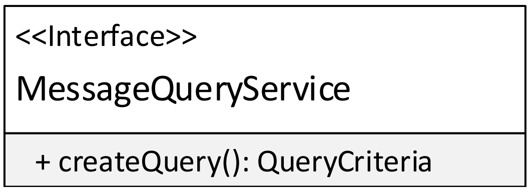


Figure 4.15: Message Query Service API

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.

4.3.3 Callback Entities

Callback entities are used by the system to interact with platform components which are not part of the middleware but that the middleware 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 4.7 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 the middleware to resolve the peers that are in a collective. This information cannot be stored in the middleware because it changes frequently, two consecutive calls might not result in the same response.
NotificationCallback	This Notification Callback is used by the middleware to notify a platform component about messages that are not intended to be handled by the middleware. This includes messages like task results or task-related information like communication errors.

Table 4.7: Callback entities of the middleware.

Peer Authentication Callback API

This callback is used to authenticate a peer within the middleware 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.

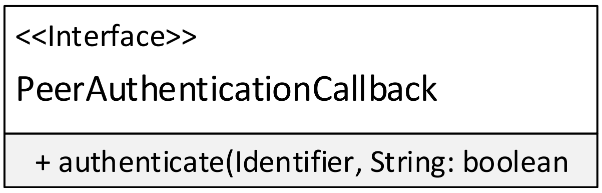

<<Interface>>
PeerAuthenticationCallback
+ authenticate(Identifier, String: boolean

Figure 4.16: Peer Authentication Callback API

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

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.

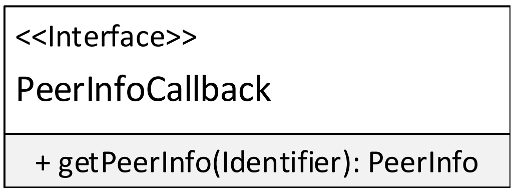


Figure 4.17: Peer Info Callback API

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 the middleware, in order for the middleware to allow to platform components the functionality of addressing their messages on the collective level.

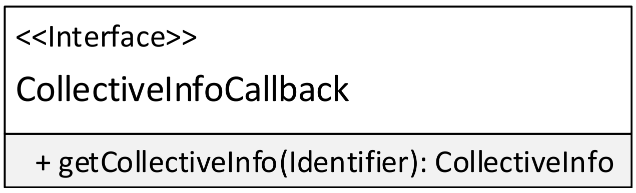


Figure 4.18: Collective Info Callback API

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 the middleware 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.

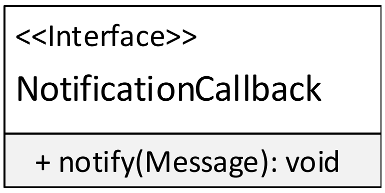


Figure 4.19: Notification Callback API

public void **notify**(Message message)

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

Parameters

message - The received message.

4.4 Algorithms

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

4.4.1 Creation of Output Adapters

Algorithm 4.1 describes how Output Adapters are created/instantiated in the Adapter Manager (see Section 4.1.2) 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.

4.4.2 Handling of Messages

Messages are handled by the Messaging and Routing Manager (see Section 4.1.2). Every incoming message, regardless of whether it is from an internal component, an application or a peer is handled by the *handleMessage* function. Algorithm 4.2 depicts the function. Line 7 of the algorithm indicates the application of a routing rule which has been described in Section 4.2.2.

First, the message is assigned with a unique message Id which is used to track the message within the middleware. 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 4.1.2). 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

```
Function createAdapterInstances is
    input  : Peer information (peerInfo)
    output: Identifiers of the created adapters
    addresses = peerInfo.addresses;
    policy = peerInfo.deliveryPolicy;
    for all address in addresses do
        if there is a stateless adapter instance available for this address then
            add the address of the stateless instance to the result list;
            if policy == PREFERRED then
                return result list;
            else
                continue with next address;
            end
        else if there is a stateful adapter implementation for this address then
            if there is already an instance of that adapter for this peer then
                add the address to the result list;
                if policy == PREFERRED then
                    return result list;
                else
                    continue with next address;
                end
            end
            instantiate new stateful adapter with a unique ID;
            add new instance to the instances of stateful adapters;
            add new instance to the result list;
            if policy == PREFERRED then
                return result list;
            else
                continue with next address;
            end
        else
            log that there was an unknown adapter;
    end
end
```

Algorithm 4.1: Creation of adapters instances for a peer based on peer's delivery policy.

```
Function handleMessage is
    input : Message (msg)
    if message Id is empty then
        create unique message ID;
    end
    createPolicyHandlers = false;
    if message receiver is not null then
        add the message receiver to the receiver list;
        createPolicyHandlers = true;
    end
    /* check if there are further receivers */
    get further receivers from the routing engine (based on routing rules);
    add them to the receiver list;
    if receiver list is empty then
        send error message to NotificationCallback;
        return;
    end
    for each receiver in receiver list do
        if receiver is component then
            forward message to component;
            continue;
        else if receiver is collective then
            deliverToCollective(msg, receiver, createPolicyHandlers); // Alg. 4.3
        else
            try
                deliverToPeer(msg, receiver, createPolicyHandlers); // Alg. 4.4
            catch
                send error message to the sender of the message;
        createPolicyHandlers = false;
    end
end
```

Algorithm 4.2: Handling of messages in the Messaging and Routing Manager.

following section. Finally the presented algorithm forwards the message to the corresponding receivers.

Algorithm 4.3 describes how the messages are forwarded to a collective and Algorithm 4.4 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 4.1.2).

```
Function deliverToCollective is
    input : Receiver (collective)
            Message (msg)
            Boolean value whether to create delivery policy handler (createHandlers)
    retrieve collective info (collInfo) from CollectiveInfoCallback;
    if createHandlers then
        // to trace the enforcement of delivery policies
        registerCollectiveMessageDeliveryAttempt(msg, collInfo.deliveryPolicy);
    end
    for each peer in collInfo.peers do
        try
            deliverToPeer(msg, receiver, createHandlers);        // see Alg. 4.4
        catch
            enforceCollectiveDeliveryPolicy(new error message);
            if collInfo.deliveryPolicy is TO_ALL_MEMBERS then
                /* delivery failed because the massage could not
                   be sent to everyone                              */
                break;
            end
    end
end
```

Algorithm 4.3: Sending messages to a collective.

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 a peer at the

```
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 4.4: Sending messages to peers.

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 4.1.2 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 4.3 and 4.4) but the enforcement of policies is handled upon reception of acknowledge and communication error messages which are sent by adapters.

Table 4.8 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 4.8: Data structure to enforce collective delivery policies (CollPolEDS). Underlined entries indicate the composed key for each entry.

Table 4.9 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 4.9: 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 behaviour can be observed in Algorithm 4.5 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 4.6 describes the delivery policy enforcement on the collective level.

```
Function enforcePeerDeliveryPolicy is
    input : Acknowledge or communication error Message (msg)
    try
        if checkPeerDeliveryPolicy(msg) then    // might throw an exception
            entry = discardPeerPolicyEntry(msg);
            if entry == null then     // Policy has already been enforced
                return;
            end
            if collectiveDeliveryPolicyHasEntry(msg) then
                enforceCollectiveDeliveryPolicy(msg);          // see Alg.  4.6
            else if entry.messagePolicy == ACKNOWLEDGE then
                send acknowledgement to entry.sender;
        end
    catch
        /* msg can only be a communication error message     */
        entry = discardPeerPolicyEntry(msg);
        if entry == null then        // Policy has already been enforced
            return;
        end
        if collectiveDeliveryPolicyHasEntry(msg) then
            enforceCollectiveDeliveryPolicy(msg);              // see Alg.  4.6
        else
            send communication error message to entry.sender;
end

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

Algorithm 4.5: Enforcing a delivery policy on the peer level.

```
Function enforceCollectiveDeliveryPolicy is
    input  : Acknowledge or communication error Message (msg)
    try
        if checkCollectiveDeliveryPolicy(msg) then            // might throw an exception
            entry = deleteCollectivePolicyEntry(msg);
            if entry.policy == ACKNOWLEDGE then
                send acknowledgement to the entry.sender;
            end
        end
    catch
        entry = deleteCollectivePolicyEntries();
        send error message to the entry.sender;
end

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

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

Algorithm 4.6: Enforcing a delivery policy on the collective level.

CHAPTER 5

Implementation and Evaluation

The first section of this chapter discusses the implementation of the proposed middleware of Chapter 4. It discusses the chosen technology and the available adapters, furthermore it describes how to extend the system with further adapters, which is one of the most important aspects of the system regarding extendibility. Section 5.1.1 analyzes the design of the middleware that has been presented in Chapter 4 regarding the requirements on collaboration patterns and requirements of HDA-CASs which have been formulated in the introduction of this thesis (Chapter 1).

The sections afterwards evaluates the approach presented in the previous chapter in terms of semantics, functionality and performance. Section 5.2 evaluates the semantics of the proposed system based on the motivating scenario presented in Section 1.2. Section 5.3 takes a look at the evaluation of the functionality. Finally this chapter discusses the performance of the prototype in Section 5.4.

5.1 Implementation

A working prototype of the middleware has been implemented in Java 1.7[1] and is available on GitHub[2]. It can be used directly by HDA-CAS platforms running on the Java Virtual Machine. Additionally, other platforms can interact with the middleware using the set of provided APIs.

The prototype implementation provides the following adapters to test the functionality of the system:

- REST input and output adapters
- SOAP input and output adapters
- Email input and output adapters

[1]James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. 2013. The Java Language Specification, Java SE 7 Edition (1st ed.). Addison-Wesley Professional.

[2]`https://github.com/tuwiendsg/SmartCom`

- Android[3] output adapter
- Dropbox[4] input and output adapters

Further information on the usage of the provided adapters can be found on GitHub[5]. Additional third-party adapters can be loaded as plug-ins and instantiated when needed.

The middleware uses MongoDB[6] as a database system for its various subsystems. Depending on the usage of the middleware, either an in-memory or dedicated database instances of MongoDB can be used. For the in-memory MongoDB instances the middleware uses the Flapdoodle Embedded MongoDB[7]. The database system could be easily changed by replacing the implementations for the various Data Access Objects (DAOs) with implementations for other database systems.

To decouple the execution of the HDA-CAS platform and the communication we use Apache ActiveMQ[8] as the Message Broker with an additional internal abstraction layer on top of the Message Broker.

The various subsystems and the whole system can be build using Apache Maven[9]. The APIs are provided in the 'api' module of the prototype implementation. The communication middleware can be started by instantiating the class *at.ac.tuwien.dsg.smartcom.SmartCom* and calling its *initializeSmartCom()* method. The methods *getCommunication()*, *getMessageInfoService()*, and *getQueryService()* provide the services and classes that allow the interaction with the middleware. Further information about the implementation can be found on GitHub[5].

5.1.1 Requirement Verification

The following section discusses how the collaboration patterns – that were formulated in Section 1.1 – are fulfilled by the suggested solution.

- *Relationship topology* – since such a relationship topology is application specific, the middleware does not handle them but also does not restrict them. By providing proper routing rules, delivery policies, and assembled collectives all kind of topologies are realizable for applications.
- *Collaboration environment* – by using adapters, the middleware is capable to use any collaboration environment that complies with the basic interaction procedure of the system.
- *Communication channels* – the suggested middleware supports all kind of communication channels by implementing a specific adapter for such a channel. This allows the system to provide extendibility and flexibility regarding future technology.

[3] `http://www.android.com/`
[4] `https://www.dropbox.com/`
[5] `https://github.com/tuwiendsg/SmartCom/wiki`
[6] `http://www.mongodb.org`
[7] `https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo`
[8] `http://activemq.apache.org`
[9] `http://maven.apache.org/`

- *Security and privacy policies* – Security functionality is provided by the Authentication Manager (see Section 4.1.2) and privacy policies (also in Section 4.1.2) are executed per service unit when messages are sent.
- *Delivery policies* – Delivery policies have been introduced in Section 4.1.2 and can be specified on the message, peer and collective level to provide the required flexibility.

The following presents how the requirements of HDA-CASs that were formulated in Section 1.1 are fulfilled by the suggested solution:

1. **Virtualization**— is achieved by introducing a layer of abstraction (i.e., the whole middleware) around the actual communication . Using an identifier for a service unit (regardless whether it is a human-based or machine-based service unit) is sufficient to send a message from the platform to a single service unit or a collective.
2. **Heterogeneity**— since the communication with service units is virtualized and service unit specific communication is handled solely within the middleware, heterogeneity is provided to the system. By adding corresponding adapters, the system is able to communicate with any kind of service unit.
3. **Communication**— the actual communication of the middleware with service units is handled by adapters.
4. **Persistence**— each message is persisted to a database by the Message Logging Service and exposed by the Message Query Service.
5. **Scalability**— to support scalability, the whole system uses message queues wherever possible and required. Scalability is achieved by scaling out of adapters to handle big workload and to scale in again if the workload is reduced to save resources. The same procedure is used by internal components that are listening on message queues. The architecture also supports that middleware components are distributed over multiple machines by utilizing the Message Broker to distribute the workload.

5.1.2 Extending the System with Adapters

This section describes how to extend the system by adding new adapters. First, we examine how to implement and register Output Adapters and afterwards how to add Input Adapters. Further information on adapters can be found in Section 4.1.1

Output Adapters

In order to create an Output Adapter that can send messages to the peers, the adapter has to implement the `OutputAdapter` interface. For the purpose of demonstration the following description focuses on a communication using SMS, any other form of communication (e.g., a dedicated mobile application, email, instant messaging) can be implemented in a similar way. 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 are 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 Output Adapter.

```java
@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 5.1: Implementation of a Output Adapter

Input Adapters

As already discussed, there are two different types of Input Adapters available. Pull adapters actively issue a pull in a certain time interval (see Listing 5.2), whereas push adapters wait for push notifications over communication channels (see Listing 5.3).

Note that both adapters have to be provided with the concrete parameters (e.g.., the url of the FTP server and credentials). This information can either be hard-coded or provided during the creation of the adapter by an application.

```java
private class FTPAdapter implements InputPullAdapter {

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

  @Override
  public Message pull() {
    //perform pull and transform to message
    return msg;
  }
}
```

Listing 5.2: Implementation of an Input Pull Adapter

```
private class MailinglistAdapter extends InputPushAdapterImpl {

  private MailinglistAdapter(...) {
    //initialize adapter
  }

  @Override
  public void init() {
    //initialize some handler

    //alternatively do the following:
    schedule(new PushTask() {

      @Override
      public void run() {
        //wait for message and transform to internal message
        publishMessage(msg);
      }
    });
  }

  @Override
  public void cleanUp() {
    //destroy resources
  }
}
```

Listing 5.3: Implementation of an Input Push Adapter

5.1.3 Interacting with the System

This section takes a brief look at how to use the implementation of the middleware in a program in the traditional sense of a 'Hello World' program.

```
public class HelloWorld implements NotificationCallback {

  public void helloWorld() throws Exception {
        //initialize peerAuthenticationCallback
        //initialize peerInfoCallback
        //initialize collectiveInfoCallback

        //add a peer with id 'peer1' that can receive emails

        //initialize smartcom:
        SmartCom smartCom = new SmartComBuilder(
          peerAuthenticationCallback,
          peerInfoCallback,
          collectiveInfoCallback)
          .create();

        //get communication API
        Communication communication = smartCom
          .getCommunication();

        //register the notification callback API
        communication.registerNotificationCallback(this);

        //register the input handler (pulls every second)
        communication.addPullAdapter(
          new EmailInputAdapter(),
          1000);

        //create message
        Message.MessageBuilder builder =
          new Message.MessageBuilder()
            .setType("TASK")
            .setSubtype("REQUEST")
            .setReceiverId(Identifier.peer("peer1")
            .setSenderId(Identifier.component("DEMO"))
            .setContent("Hello␣World!");
        Message msg = builder.create();

        //send the message
        communication.send(msg);
  }

  public void notify(Message message) {
    System.out.println(
          "Received:"+
          message.toString();
  }
}
```

Listing 5.4: Hello World using email

Listing 5.4 outlines a program that sends a ’hello world’ message to a peer by email. Email is used for the communication because an external peer is registered in the system that provides an email address in his peer profile, which can be accessed by the peerInfoCallback (external to the middleware). Additionally an Input Adapter for email is registered, that listens for incoming emails on a specific account. When the peer receives the ’hello world’ email, it can send a response to the email account monitored by the Input Adapter. Since the Input Adapter is

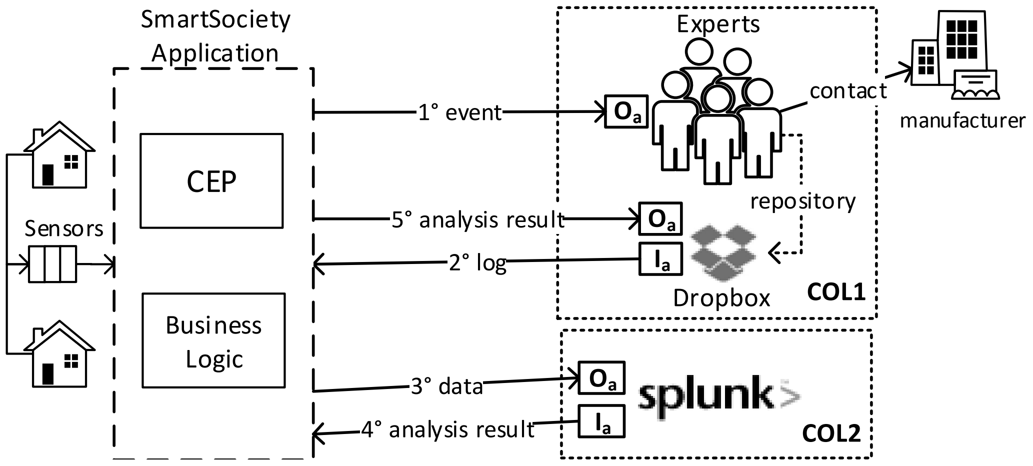


Figure 5.1: Supporting predictive maintenance use-case. Collectives of human expert and software service units participate in a joint collaboration to identify the cause of a detected malfunction event.

executed every second, it detects at some point, that a new email arrived, which is consequently transformed to a message and forwarded to the middleware. Finally, this response message is sent by the middleware to the Notification Callback API, which is implemented by the HelloWorld class. The complete 'hello world' code is available on GitHub[10].

5.2 Requirement and Design Evaluation

The following section formulates a concrete use-case to validate the presented design and its fulfilment of the stated requirements based on the second motivating scenario that has been presented in Section 1.2.

A predictive maintenance SmartSociety application receives sensor readings from a smart building and performs Complex Event Processing (CEP) on them. If an indication of a potential malfunction is detected, further investigation is required. A collective (COL1) of available human experts is formed[11] *and a collaborative pattern imposed. The application appoints an expert to lead the peer collaboration within the collective and sets up a Dropbox repository for sharing the findings and equipment logs between the SmartSociety application and the collective. Additionally, it provides to the COL1 manager the contact details of the manufacturer of the malfunctioning equipments in case additional consultations are required. Finally, middleware also provides COL1 peers with mediated access to a data analysis tool (e.g., Splunk*[12]*).*

Figure 5.1 shows the two collectives participating in this scenario. COL1 containing human expert service units (SUs) and a single software SU — the Dropbox service. Furthermore, each

[10] `https://github.com/tuwiendsg/SmartCom`
[11] Selection of collective peers is out of scope of this thesis.
[12] `www.splunk.com`

human SU is assigned a dedicated peer adapter (O_a) instance, while for the Dropbox service, both a O_a and a feedback adapter (I_a) instance are executed, in order to support two-way communication with the SmartSociety platform. COL2 contains a single SU that does data analysis. To support two-way communication we introduce again a O_a and a I_a.

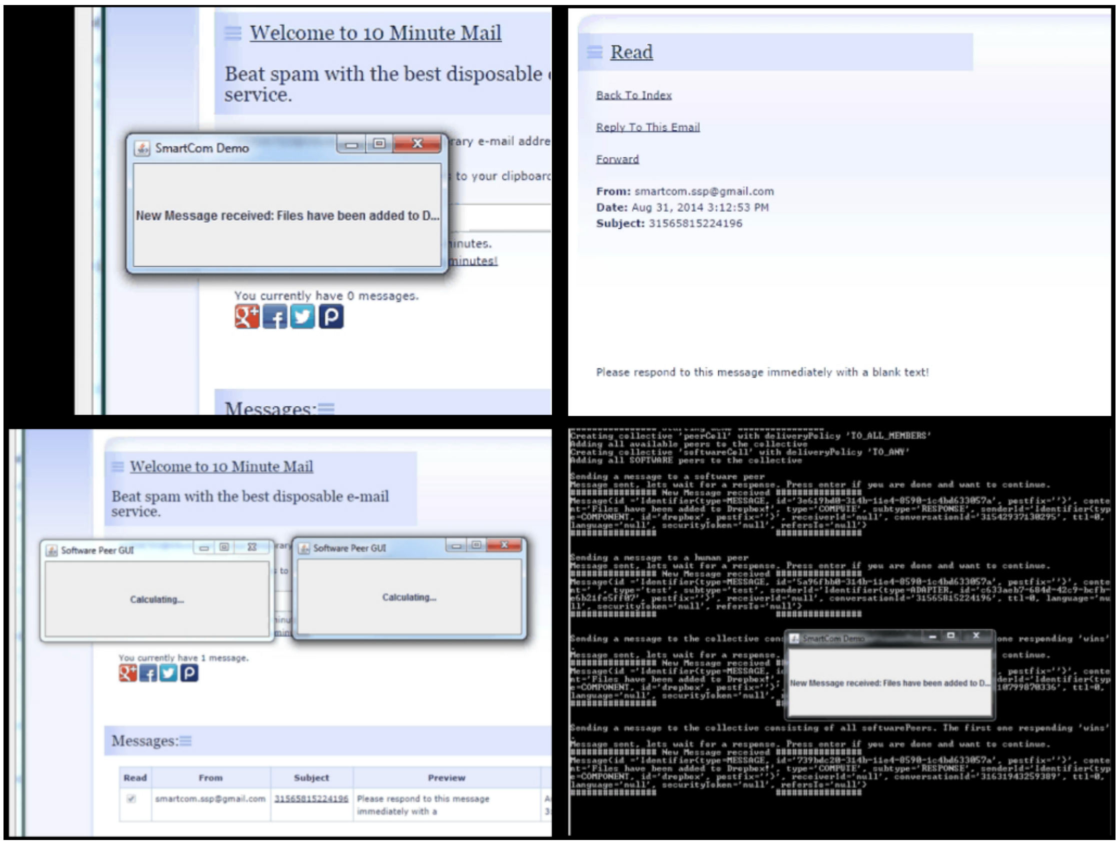
The use-case starts by SmartSociety application notifying peers that their participation is needed (Fig. 5.1, `1°`) by sending a message to `MessagingAndRoutingManager` which will initialize the routing. Some peers expressed in their profiles the preference for being notified by SMS, others by email. To send an SMS, the `MessagingAndRoutingManager` reads the phone number of a peer from its profile and hands it to `AdapterManager` which instantiates and executes the SMS adapter. The `PeerAdapter` sends the message by using the most cost-efficient mobile operator. Peers that prefer to be contacted through email will be sent an email using a stateless email adapter through an external mail service. If a peer wants to be contacted by email and SMS, it can set this preference by using the `DeliveryPolicy`. The content of the message is provided by the SmartSociety application. In this case, the message contains the URL pointing to the description of the detected event, Dropbox repository URL and access tokens for sharing the results, the name and contact details of the selected collective manager as well as a natural language description of the required activities and contractual terms. Furthermore, the manager is sent the contact details of the equipment manufacturer's customer service, and the address of another collective – COL2, which in practice contains a single software peer, the Splunk service.

For the sake of simplicity, we assume that expert peers do accept the terms and participate in COL1. The manager freely organizes the collaboration in COL1. At a certain point, human peers need to run an additional data analysis on the log. The collaboration pattern foresees that if a file with predefined filename is deposited in the shared Dropbox repository, the dedicated feedback adapter would pick up that file (`2°`) and forward it to the COL2 for analysis. The middleware ensures that `FeedbackPullAdapter` for Dropbox `(DropboxFeedbackAdapter)` regularly checks if there are new files available (e.g., once a minute). The system will then create and send a message to the Splunk Peer Adapter which contains the location of the file and further information on the analysis (`3°`). Once Splunk has finished analyzing the data, Splunk will deposit the results file back to the Dropbox repository (`4°` + `5°`) and its `FeedbackPushAdapter` will push a multicast notification message to the COL1 members (`1°` again). The COL1 can then continue their work.

5.3 Functional Evaluation

A video that demonstrates the functionality of sending messages to and receiving messages from single service units, homogeneous and heterogeneous collectives can be observed on GitHub[13]. This demo presents the use cases of sending a message to a software peer, to a human peer, to a homogeneous collective (only software peers) and to a heterogeneous collective (software and human peers). The demo setup is the following: two software peers which are available via REST and which upload files to the Dropbox, and two human peers which can be contacted

[13]`https://raw.githubusercontent.com/wiki/tuwiendsg/SmartCom/demo.avi`


Welcome to 10 Minute Mail
Beat spam with the best disposable
service.
SmartCom Demo
New Message received: Files have been added to D...
You currently have 0 messages.
Messages:
Read
Back To Index
Reply To This Email
Forward

From: smartcom.ssp@gmail.com
Date: Aug 31, 2014 3:12:53 PM
Subject: 31565815224196

Please respond to this message immediately with a blank text!

Welcome to 10 Minute Mail
Beat spam with the best disposable e-mail service.
Software Peer GUI
Calculating...
Software Peer GUI
Calculating...
You currently have 1 message.
Messages:

Read	From	Subject	Preview
✓	smartcom.ssp@gmail.com	31565815224196	Please respond to this message immediately with a

```
Creating collective 'peerCell' with deliveryPolicy 'TO_ALL_MEMBERS'
Adding all available peers to the collective
Creating collective 'softwareCell' with deliveryPolicy 'TO_ANY'
Adding all SOFTWARE peers to the collective

Sending a message to a software peer
Message sent, lets wait for a response. Press enter if you are done and want to continue.
################## New Message received ##################
Message(id ='Identifier(type=MESSAGE, id='3e619bd0-314b-11e4-8590-1c4bd633057a', postfix='')', conte
nt='Files have been added to Dropbox!', type='COMPUTE', subtype='RESPONSE', senderId='Identifier(typ
e=COMPONENT, id='dropbox', postfix='')', receiverId='null', conversationId='31542937130295', ttl=0,
language='null', securityToken='null', refersTo='null')
##################                        ##################

Sending a message to a human peer
Message sent, lets wait for a response. Press enter if you are done and want to continue.
################## New Message received ##################
Message(id ='Identifier(type=MESSAGE, id='5a96fbb0-314b-11e4-8590-1c4bd633057a', postfix='')', conte
nt=' ', type='test', subtype='test', senderId='Identifier(type=ADAPTER, id='c633aeb7-684d-42c9-bcfb-
e6b21fe5ff07', postfix='')', receiverId='null', conversationId='31565815224196', ttl=0, language='nu
ll', securityToken='null', refersTo='null')
##################                        ##################

Sending a message to the collective cons                        one responding 'wins'
.
Message sent, lets wait for a response.                         continue.
################## New Message received ##
Message(id ='Identifier(type=MESSAGE, id                        , postfix='')', conte
nt='Files have been added to Dropbox!',                         derId='Identifier(typ
e=COMPONENT, id='dropbox', postfix='')',                        10799870336', ttl=0,
language='null', securityToken='null', r
##################                        ##

Sending a message to the collective consisting of all softwarePeers. The first one responding 'wins'
.
Message sent, lets wait for a response. Press enter if you are done and want to continue.
################## New Message received ##################
Message(id ='Identifier(type=MESSAGE, id='739bdc20-314b-11e4-8590-1c4bd633057a', postfix='')', conte
nt='Files have been added to Dropbox!', type='COMPUTE', subtype='RESPONSE', senderId='Identifier(typ
e=COMPONENT, id='dropbox', postfix='')', receiverId='null', conversationId='31631943259389', ttl=0,
language='null', securityToken='null', refersTo='null')
##################                        ##################
```

Figure 5.2: Four screenshots of the demonstration video. **Top-left**: software peer added a file to a dropbox folder which is detected by the Dropbox input adapter of the middleware. **Top-right**: human peer receives email by Email output adapter. **Bottom-left**: collective of software peers receives a message. **Bottom-right**: Several respond messages sent by the middleware using the Notification Callback API.

by email[14] and which respond by email. First, a message is sent to one of the software peers which immediately uploads a file. This is detected by the middleware and displayed in the log. Afterwards, a message is sent to a human peer which receives the message in his email client and immediately responds via email. This is also detected and displayed by the middleware. In the rest of the demonstration, collective capabilities are demonstrated. First, a collective of both software peers is formed and a REST call is made to both peers, which immediately respond. Afterwards, a collective of all available peers (two software peers and two human peers) is formed and they all receive the same message via REST or email. Figure 5.2 presents four screenshots of the demo video.

Furthermore, the functionality of the proposed system has been evaluated by integrating

[14]Used email provider: 10 Minute Mail (`http://10minutemail.com/`)

it with a CAS Provisioning Service called SALAM[15] (SociAL compute unit (SCU) runtime frAmework and siMulation). In the integration demo, the purpose of SALAM is to create collectives of peers based on tasks and task requirements, and to initiate the sending of messages to the peers which describe the task. Peers are maintained by SALAM and the provisioning is based on the profiles of these peers. The code used for the integration can be found on GitHub[16].

Finally the approach has also been evaluated in terms of the SmartSociety project. It has been integrated with a Peer Manager that provides the data for the Peer Info Callback API, Collective Info Callback API, and Peer Authentication Callback API (see Section 4.3.3). Furthermore, the middleware is used by the application 'Ask SmartSociety!', which allows users to ask questions which will be solved by peers. The application has been created to demonstrate the integration of various components of the SmartSociety platform.

5.4 Performance Evaluation

The following performance evaluations of the presented prototype have been 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 configuration of the performance evaluation environment is the following:

- One implementation of a Stateless Output Adapter (one instance shared by all peers).
- Ten Input Push Adapter that 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.
- 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 meaningful data of sent/received messages.
- Only sent and received messages are considered as 'handled', no internal messages.

Figure 5.3 depicts the setup for the performance evaluation as described above.

The performance has been evaluated for every combination of 1, 5, 10, 20, 50, 100, and 1000 Workers sending messages concurrently, as well as 1, 10, 100, and 1000 registered peers. Each test run has been executed 10 times to get a meaningful number of possible throughput. Figure 5.4 presents the results of the test runs, the data can be found on GitHub[17].

As one can see, the initial throughput is around 5.000 messages per second for a lower number of peers. The limiting factor here is the used message broker which only allows approximately 22.000 messages per second [18] on a machine with the previously mentioned specifications. The system has an upper bound of approximately 5.000 messages per second since each message sent by a worker is handled multiple times by the message broker and the middleware.

[15]`https://github.com/tuwiendsg/SALAM`

[16]`https://github.com/tuwiendsg/SALAM/tree/master/smartcom-salam`

[17]`https://github.com/tuwiendsg/SmartCom/tree/master/performance`

[18]`http://activemq.apache.org/performance.html`

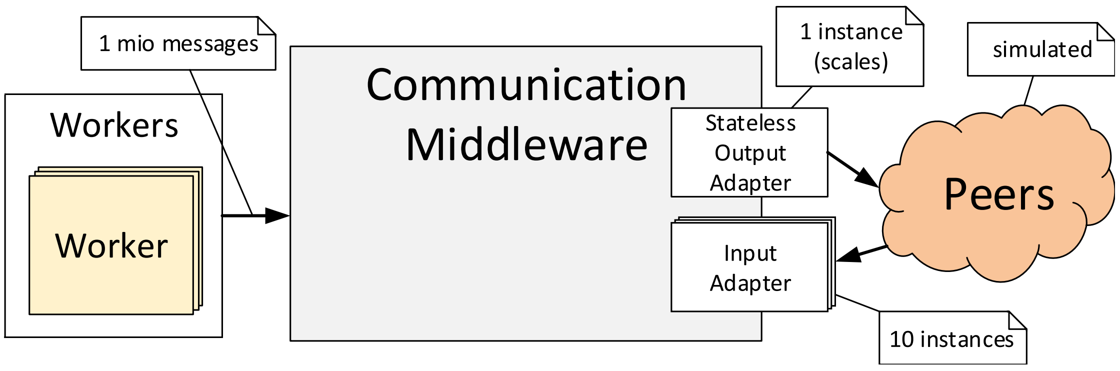


Figure 5.3: Setup for the performance evaluations.

The performance decrease with higher amounts of peers is a result of the increased memory requirements rather than computational complexity. Note that the system has been designed to be able to be distributed on various machines, which can eliminate the problem of not having enough memory on a single machine.

By looking at the data, one can observe, that the throughput is initially quite low but increases significantly during the execution. This is the result of the chosen scaling-out strategy of the Stateless Output Adapter.

Note that performance is not a primary concern of the middleware since the response times of humans are usually much larger compared to the response times of machines. Nevertheless, the performance of the system is an important aspect considering the usability and success of the middleware.

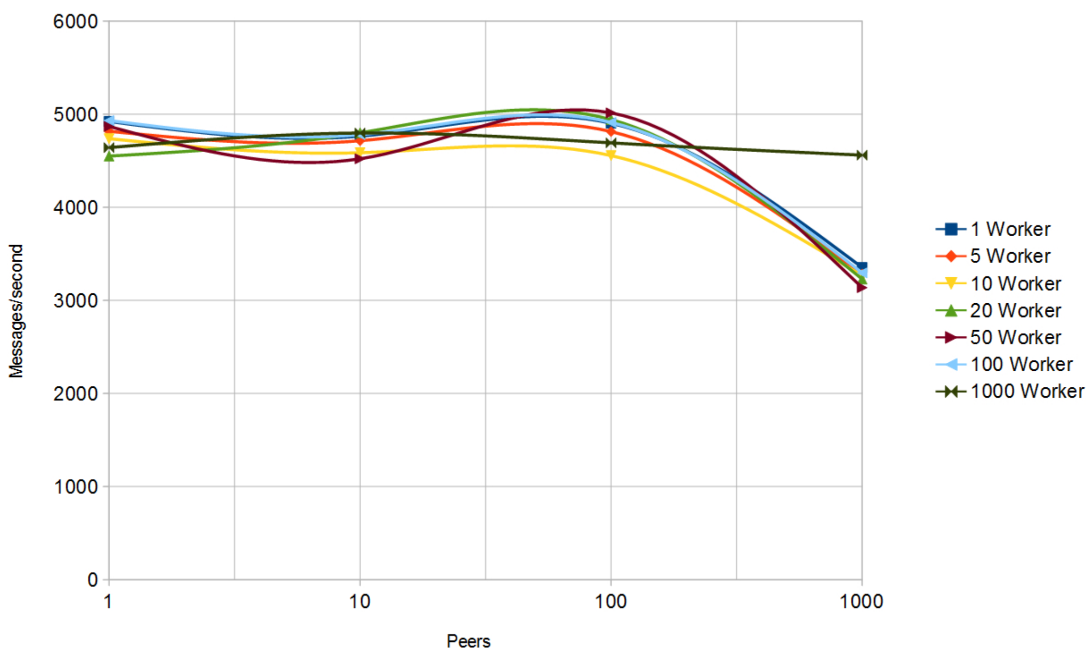


Figure 5.4: Performance Evaluation of the Middleware on: Windows 7 64-bit, Intel Core2 Duo with 2x 2.53 GHz, 4.00 GB DDR2-RAM.

CHAPTER 6

Conclusions and Future Work

6.1 Summary

This thesis presented the concepts for virtualization of communication in Hybrid and Diversity-Aware Collective Adaptive System (HDA-CAS) platforms – platforms with the fundamental properties of *Collectiveness*, *Adaptiveness* and *Hybridity* (i.e., humans, machines, and things communicate with each other). Furthermore, it introduces the architecture and design of a communication middleware for HDA-CASs which is based on the concepts presented in this thesis. The solution aims to tackle hybridity in such platforms by virtualizing the communication with human-based and machine-based service units by creating a transparent way of addressing of service units and providing technology-independent communication with them for the platform. The middleware considers a variety of aspects, including different kinds of supported service units (human- vs. machine-based service units), different channels of communication (e.g., email, SMS, and FTP), and loose-coupling to promote the usage of familiar third-party services. This is of high importance in order to create a platform which is able to scale to a potentially high number of diverse service units organized in multiple dynamic collectives without restricting self-organization and interaction of service units.

To support collectiveness, the middleware allows addressing collectives of service units transparently to the HDA-CAS, relieving the HDA-CAS programmer the duty to keep track of current members of a collective, allowing the collective to scale up and down when needed seamlessly. The described system was validated through a prototype implementation provided as open-source.

6.2 Future Work

The focus of future work is to improve the presented prototype and integrate it into the SmartSociety platform. Especially, the improvement of the scalability and the overall performance of the middleware will be important for the success of the platform adoptions. The performance analysis in Section 5.4 discusses the problems of the current solution. Furthermore, the middleware has to be extended to provide more flexibility, and easier and more advanced configuration possibilities (e.g., configuration of thread pools for the various components).

An important aspect will also be the improvement of the Message Info Service to provide machine- and human-readable descriptions. Providing the scientific fundamentals and the implementation of such a system is, due to the complexity, out of context of this thesis. Adding a translation service for human-readable messages to machine-readable messages would also be possible and might further improve the applicability of the middleware.

The focus of further research will be on modeling the primitives for integrated monitoring and execution of elasticity actions, such as imposing of optimal topologies, dynamical adjustment of collective members, and support for incentive application. Currently, these actions have to be fully specified on the HDA-CAS application level, presenting an unnecessary burden for the developers.

List of Terms and Acronyms

adapter

An adapter is used by the middleware to abstract the communication with peers and to handle the communication technology independently within the system. Adapters are described in detail in Section4.1.1 and Section 3.2.2

API

application programming interface

application

An application that runs on the HDA-CAS and uses the middleware to communicate with peers.

CAS

Collective Adaptive System

collective

A collective is a group of peers that have been assembled to provide a common service (e.g., data analysis). This groups consist of at least one peers and can change at any time.

communication channel

A communication channel indicates a type of technology related communication with a peer. For example an email or a REST call. Each adapter handles the communication over a specific communication channel.

HDA-CAS

Hybrid and Diversity-Aware Collective Adaptive System

peer

A peer is either a human or a machine and provides a service.

platform

HDCAS platform that uses the middleware

platform component

Component of the HDA-CAS that interacts with the middleware.

service unit

Entity that consists of a peer (human, machine, or things) and a context (the concept of service units will be described in detail in Section 3.2.1). Can be human-based or machine-based service unit.

tool

External tools that are used by peers to finish tasks (e.g., by putting a file on a file server) or interact with the system.

Bibliography

[1] Amazon Mechanical Turk. `http://www.mturk.com`. Accessed: 2014-11-30.

[2] Applications of the SmartSociety project. `http://www.smart-society-project.eu/applications/`. Accessed: 2014-11-30.

[3] The Foundation for Intelligent Physical Agents. `http://fipa.org/`. Accessed: 2014-11-20.

[4] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In Erol Şahin and WilliamM. Spears, editors, *Swarm Robotics*, volume 3342 of *Lecture Notes in Computer Science*, pages 10–20. Springer Berlin Heidelberg, 2005.

[5] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. The jabberwocky programming environment for structured social computing. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 53–64. ACM, 2011.

[6] Vasilios Andrikopoulos, Antonio Bucchiarone, Santiago Gómez Sáez, Dimka Karastoyanova, and Claudio Antares Mezzina. Towards modeling and execution of collective adaptive systems. In *Service-Oriented Computing–ICSOC 2013 Workshops*, pages 69–81. Springer, 2014.

[7] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[8] Antonio Bucchiarone, Alberto Lluch Lafuente, Annapaola Marconi, and Marco Pistore. A formalisation of adaptable pervasive flows. In *Proceedings of the 6th International Conference on Web Services and Formal Methods*, WS-FM'09, pages 61–75, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001*, pages 316–330. Springer, 2001.

[10] Giacomo Cabri, Elton Domnori, and Davide Orlandini. Implementing agent interoperability between language-heterogeneous platforms. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, pages 29–34. IEEE, 2011.

[11] Nicola Capodieci and Giacomo Cabri. Collaboration in swarm robotics: A visual communication approach. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 195–202. IEEE, 2013.

[12] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[13] Tiziana Catarci, Massimiliano de Leoni, Andrea Marrella, Massimo Mecella, Berardino Salvatore, Guido Vetere, Schahram Dustdar, Lukasz Juszczyk, Atif Manzoor, and Hong-Linh Truong. Pervasive software environments for supporting disaster responses. *Internet Computing, IEEE*, 12(1):26–37, 2008.

[14] Brahim Chaib-draa and Frank Dignum. Trends in agent communication language. *Computational intelligence*, 18(2):89–101, 2002.

[15] David Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[16] 4CaaSt consortium. Immigrant paas technologies: Scientific and technical report. `http://4caast.morfeo-project.org/wp-content/uploads/2011/02/D7.1.1-M14-PU-Immigrant-PaaS-technologies-Scientific-and-technical-report.pdf`, 2011.

[17] Massimiliano de Leoni, Fabio De Rosa, Andrea Marrella, Massimo Mecella, Antonella Poggi, Alenka Krek, and Francesco Manti. Emergency management: from user requirements to a flexible p2p architecture. In *Proc. 4th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2007)*, 2007.

[18] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. Bpel4chor: Extending bpel for modeling choreographies. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 296–303. IEEE, 2007.

[19] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, 2001.

[20] S. Anderson et al. *FoCAS Book: Adaptive Collective Systems – Herding Black Sheep*. Open publication, 2013.

[21] Tim Finin, Jay Weber, Gio Wiederhold, Mike Genesereth, Rich Fritzson, Don McKay, Stu Shapiro, Jim McGuire, Richard Pelavin, and Chris Beck. Specification of the kqml agent-communication language. 1994.

[22] FIPA. Fipa'97 specification part 2: Acl. 1997.

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

[24] Apache Software foundation. Apache ServiceMix. `http://servicemix.apache.org/`. [Online; accessed September 2014].

[25] Fausto Giunchiglia, Vincenzo Maltese, Stuart Anderson, and Daniele Miorandi. Towards hybrid and diversity-aware collective adaptive systems. 2013.

[26] Maria Golemati, Akrivi Katifori, Costas Vassilakis, George Lepouras, and Constantin Halatsis. Creating an ontology for the user profile: Method and applications. In *Proceedings of the First RCIS Conference*, pages 407–412, 2007.

[27] Object Management Group. Common Object Request Broker Architecture (CORBA) Specification Version 3.3, 2012.

[28] Karen Henricksen, Jadwiga Indulska, Ted McFadden, and Sasitharan Balasubramaniam. Middleware for distributed context-aware systems. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 846–863. Springer, 2005.

[29] WSO2 Inc. WSO2 Enterprise Service Bus. `http://wso2.com/products/enterprise-service-bus/`. [Online; accessed September 2014].

[30] Red Hat JBoss. JBoss ESB. `http://jbossesb.jboss.org/`. [Online; accessed September 2014].

[31] Daniela Kengyel, Ronald Thenius, Karl Crailsheim, and Thomas Schmickl. Influence of a social gradient on a swarm of agents controlled by the beeclust algorithm. In *Advances in Artificial Life, ECAL*, volume 12, pages 1041–1048, 2013.

[32] Serge Kernbach, Thomas Schmickl, and Jon Timmis. Collective adaptive systems: Challenges beyond evolvability. *arXiv preprint arXiv:1108.5643*, 2011.

[33] Yannis Labrou, Tim Finin, and Yun Peng. The current landscape of agent communication languages. *IEEE Intelligent systems*, 14(2):45–52, 1999.

[34] Jean-Jacques Laffont and David Martimort. *The theory of incentives: the principal-agent model*. Princeton University Press, 2009.

[35] Greg Little, Lydia B Chilton, Max Goldman, and Robert C Miller. Turkit: tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD workshop on human computation*, pages 29–30. ACM, 2009.

[36] Philip Mayer, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Toma Bure. The autonomic cloud: a vision of voluntary, peer-2-peer cloud computing. In *Self-Adaptation and Self-Organizing Systems Workshops (SASOW), 2013 IEEE 7th International Conference on*, pages 89–94. IEEE, 2013.

[37] Patrick Minder and Abraham Bernstein. CrowdLang: programming human computation systems. Technical report, University of Zurich, 2012.

[38] D Miorandi, V Maltese, M Rovatsos, A Nijholt, and J Stewart. Social collective intelligence: Combining the powers of humans and machines to build a smarter society, 2014.

[39] MuleSoft. Mule ESB. `http://www.mulesoft.com/platform/soa/mule-esb-open-source-esb`. [Online; accessed September 2014].

[40] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE pervasive computing*, 1(4):74–83, 2002.

[41] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[42] OASIS Committee Specification. Ws-bpel extension for people (bpel4people) specification version 1.1. `http://docs.oasis-open.org/bpel4people/bpel4people-1.1.html`, 2010.

[43] Hong-Linh Truong, Schahram Dustdar, and Kamal Bhattacharya. Conceptualizing and programming hybrid services in the cloud. *International Journal of Cooperative Information Systems*, 22(04), 2013.

[44] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[45] Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, pages 108–113. IEEE, 2011.

[46] Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. On Self-adaptation, Self-expression, and Self-awareness in Autonomic Service Component Ensembles. In *2011 International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, Ann Arbor (MC), October 2011. IEEE CS Press.

[47] Philipp Zeppezauer, Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Virtualizing communication for hybrid and diversity-aware collective adaptive systems. In *Proceedings of the 10th International Workshop on Engineering Service-Oriented Applications (WESOA'14), 12th International Conference on Service Oriented Computing*, Paris, France, November 2014.