

Master's Thesis

**A loosely coupled peer-to-peer  
workflow system**

carried out at the

Information Systems Institute  
Distributed Systems Group  
Technical University of Vienna

under the guidance of

Ao.Univ.Prof. Dr. Schahram Dustdar  
and

Ao.Univ.Prof. Dr. Harald Gall  
as the contributing advisor responsible

by

Daniel Schwarz  
Haberlgasse 30/9, 1160 Vienna  
Matr.Nr. 9551015

Vienna, 12 May 2004

---

To my parents

## **Acknowledgments**

First I want to thank my advisors Schahram Dustdar and Harald Gall who supported me during the whole work.

I also want to thank my mother Karoline and my father Karl for their support during the last years. With their patience and support they made it possible for me to finish my studies and write this master thesis.

## **Abstract**

In this thesis we describe a workflow system which is built on top of an existing peer-to-peer architecture. The goal of this system is to implement a framework which provides the basic infrastructure for flow-controlled and distributed human workflow [1], supported with the service oriented workflow language BPEL. The domain of the system primarily targets the execution of formally defined business processes. In order to have parts of the process distributed to different peers, a coordinator is responsible for choosing them and assigning the work. The election of the peers is performed using certain quality criteria which have to be provided by the participants of the workflow system.

A main focus in the choice of the used infrastructure as well as in the modeling of the communication structure was to comply with the increasing diversity of network clients, which range from mobile devices to personal computers. The information exchange between connected peers has to serve the requirements of both permanent network connectivity and limited availability. This is achieved by a "loosely coupled" approach: The clients gather information in connected state and maintain a local knowledge database which provides necessary information if the peer is disconnected from the network.

## Zusammenfassung

In der hier vorgestellten Diplomarbeit wird ein auf eine vorhandene Peer-to-Peer-Architektur aufbauendes Workflow-System beschrieben. Ziel dieses Systems ist es, ein Framework zu implementieren, welches eine Basis für ablaufgesteuerten "human workflow" liefert. Das Aufgabengebiet dieser Anwendungen erstreckt sich vorrangig auf das Abhandeln klar definierter Geschäftsprozesse, welche in der Sprache BPEL beschrieben sind. Dabei wird von einem Koordinator versucht, die Ausführung einzelner Prozessteile zwischen verschiedenen Peers aufzuteilen. Die Auswahl dieser erfolgt unter Zuhilfenahme von Qualitätskriterien, welche von den einzelnen Peers zur Verfügung gestellt werden.

Ein Hauptaugenmerk bei der Wahl der verwendeten Infrastruktur und bei der Modellierung der Kommunikationsstruktur war es, der immer stärker werdenden Präsenz mobiler Endgeräte ebenso Rechnung zu tragen wie gewöhnlichen PC's. Dabei wurde versucht, beim Datenaustausch zwischen den vernetzten Peers eine vernünftige Balance zwischen limitierter Verbindungsverfügbarkeit und permanenter Netzanbindung zu finden. Der Ansatz "loosely coupled" wird dadurch erreicht, dass mittels lokalen Wissensdatenbanken auf den Endgeräten auch im Offlinebetrieb der zuletzt aktuelle Status des Netzwerks verfügbar ist und beim Wiederverbinden die nötigen Daten synchronisiert werden.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Organization of this thesis . . . . .	1
1.2	Illustrative Example . . . . .	2
1.3	Problem Definition . . . . .	4
<b>2</b>	<b>Used Technology</b>	<b>6</b>
2.1	Peer to Peer . . . . .	6
2.1.1	PeerWare . . . . .	7
2.1.2	MOTION . . . . .	11
2.2	Workflow concept . . . . .	16
2.3	Description Language . . . . .	17
2.3.1	BPEL . . . . .	17
2.4	XML Data Manipulation . . . . .	18
2.4.1	The PDOM Component . . . . .	18
2.4.2	The XQL Processor . . . . .	18
2.5	Related Work . . . . .	20
2.5.1	JXTA . . . . .	20
2.5.2	Web Workflow Peers . . . . .	21
2.5.3	Serviceflow . . . . .	24
<b>3</b>	<b>Conceptual Design</b>	<b>27</b>
3.1	The scenario . . . . .	27
3.2	Software Architecture . . . . .	28
3.2.1	Workflow Coordinator . . . . .	28
3.2.2	Workflow Database . . . . .	29
3.3	Communities . . . . .	30
3.4	Processes, Tasks, and Instances . . . . .	32
3.4.1	Process . . . . .	32
3.4.2	Task . . . . .	32
3.4.3	Instance . . . . .	35
3.5	Quality of Service . . . . .	37
3.6	Data distribution . . . . .	38
3.6.1	Messages . . . . .	38

3.6.2	Data Files . . . . .	39
3.7	Instance lifecycle . . . . .	40
3.8	Task status . . . . .	41
<b>4</b>	<b>Use cases</b>	<b>43</b>
4.1	Process Management . . . . .	44
4.2	Task Negotiation . . . . .	46
4.3	Instance Execution . . . . .	48
<b>5</b>	<b>Implementation</b>	<b>51</b>
5.1	Class diagram . . . . .	51
5.1.1	WFCoordinator . . . . .	51
5.1.2	WFDatabase . . . . .	53
5.1.3	WFTask, WFData, WFTaskQoS . . . . .	54
5.1.4	WFUser . . . . .	54
5.1.5	WFInstance . . . . .	55
5.1.6	WFInstanceTask . . . . .	55
5.1.7	WFEventHadler . . . . .	55
5.2	Process description and BPEL . . . . .	56
5.2.1	Structure . . . . .	57
5.3	Internal database format . . . . .	60
5.4	Peer messaging . . . . .	62
5.4.1	Announce a new community . . . . .	62
5.4.2	Remove a community . . . . .	62
5.4.3	Global search for a task . . . . .	63
5.4.4	Announce a task . . . . .	63
5.4.5	Revoke a task announcement . . . . .	64
5.4.6	Request the process description for a task . . . . .	64
5.4.7	Send process description . . . . .	64
5.4.8	Request a task . . . . .	65
5.4.9	Accept a task request . . . . .	65
5.4.10	Refuse a task request . . . . .	65
5.4.11	Assign a task to a peer . . . . .	66
5.4.12	Start execution of a task . . . . .	66
5.4.13	Finish execution of a task . . . . .	66
5.5	Exchange of documents . . . . .	67
5.5.1	Process description files . . . . .	67
5.5.2	Instance documents . . . . .	68
<b>6</b>	<b>Validation</b>	<b>72</b>
6.1	Process Management . . . . .	73
6.1.1	Coordinator: Add a community . . . . .	74
6.1.2	Coordinator: Add process description . . . . .	75
6.1.3	Coordinator: Add process to a community . . . . .	76

6.1.4	Coordinator: Create a process instance . . . . .	77
6.1.5	Worker: Subscribe to a process community . . . . .	77
6.1.6	Worker: Download process description . . . . .	78
6.1.7	Worker: Provide a task . . . . .	79
6.2	Task negotiation . . . . .	80
6.2.1	Coordinator: Request a task . . . . .	81
6.2.2	Worker: Accept a task . . . . .	82
6.2.3	Coordinator: Assign a task to a worker peer . . . . .	83
6.3	Instance execution without input data . . . . .	84
6.3.1	Worker: Start task . . . . .	85
6.3.2	Worker: Execute and finish task . . . . .	86
6.4	Instance execution with input data . . . . .	88
6.5	Retrieve instance output documents . . . . .	91
<b>7</b>	<b>Conclusion and outlook</b>	<b>92</b>
	<b>List of Figures</b>	<b>94</b>
	<b>Bibliography</b>	<b>96</b>

# Chapter 1

## Introduction

### 1.1 Organization of this thesis

*Chapter 1* introduces an example which demonstrates a typical application we aim to solve and gives a problem definition.

*Chapter 2* provides a technological overview of peer-to-peer middlewares. We also present a suitable language for process definitions and a slim XML database.

In *Chapter 3* the detailed ideas of our approach are presented. First we describe the software architecture and the main components. Later the concepts of process management are discussed.

*Chapter 4* shows which user interactions can be applied to the system. The use cases demonstrate the actions the users of the workflow system can take and the influence of the different roles the peers can play.

*Chapter 5* focuses on some implementation details. First we give a description of the classes and the database storage. Then we describe the message and document exchange between peers.

*Chapter 6* validates the prerequisites against the results which have been achieved.

In *Chapter 7* we outline some future extensions and enhancements of the current implementation in respect of practical issues.

## 1.2 Illustrative Example

In this section I will introduce a workflow example which is used to illustrate my work in several parts of this thesis. It shows what kind of problem we address and which we want to solve in this approach. The example is taken from the WSFL specification [2].

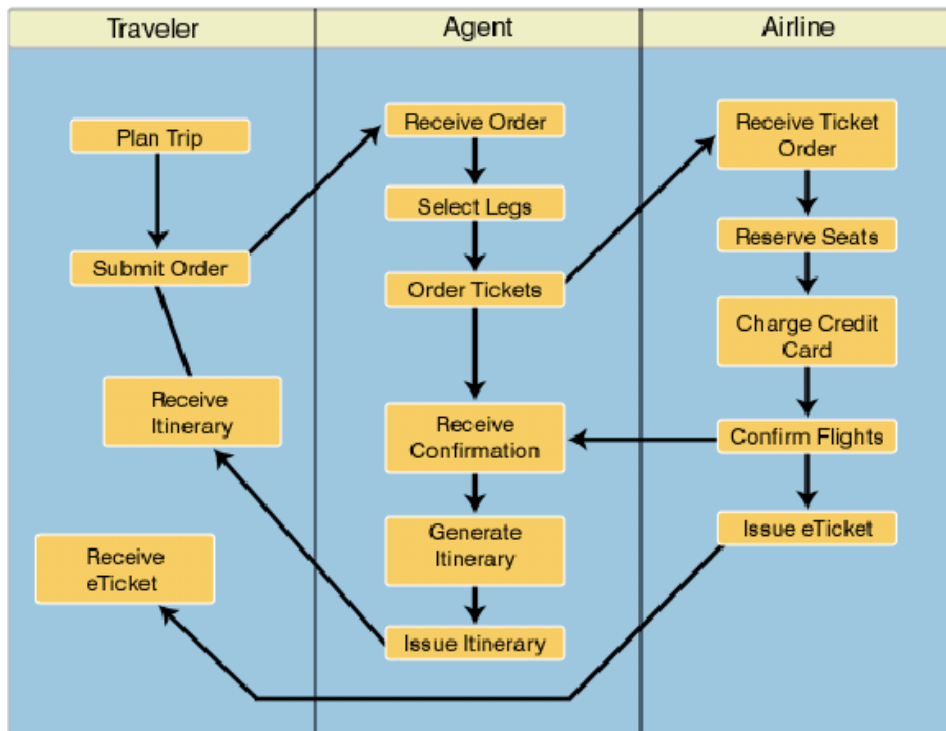


Figure 1.1: Airline reservation

A traveler plans a trip by specifying the various stages of his overall journey. For each of the stages, the traveler specifies the location that he wants to visit as well as the date when he wants to begin and the date when he wants to end the particular stay. When the traveler is finished with this, he sends this information as well as the information about the credit card to be charged for the ordered tickets to the travel agent. Next, the traveler will await the submission of the electronic tickets as well as the final itinerary for the trip.

When the agent receives the traveler's trip order, he will determine the legs for each of the stages, which includes an initial seat reservation. To actually make the corresponding ticket orders the agent submits these legs together with the information about the credit card to be charged to the airline company. Then, the agent waits for the confirmation of the flights,

which especially includes the actual seats reserved for each of the participants. This information is completed into an itinerary, which is then sent to the traveler.

When the airline receives the ticket order submitted by the agent, the requested seats will be checked and if available assigned to the traveler. After that, the credit card will be charged, and the updated leg information is sent back to the agent as confirmation of the flights. After that, the airline sends the electronic tickets to the traveler. Information about the recipient of the tickets has been specified by the traveler when instantiating the trip order process and this information is passed to the agent as well as to the airline.

### 1.3 Problem Definition

If we take a look at the workflow problem introduced in the last section, we can see, that several persons are required to carry out this type of flight reservation. We have three participants in this business process which play different roles: A customer who wants to book a flight, an agent who processes the inquiry and an airline who does the effective booking and seat reservation. These parties work autonomously and only exchange data between them on demand.

Using a peer-to-peer network as basis of the communication between the participants, we face several problems, which have to be solved in order to guarantee a decent execution of a business process. The following issues have to be considered:

- Within a peer-to-peer network we have an undefined number of peers which are potential partners in the execution of a business process. Peers can join and leave the network at any time without giving notice to other peers.
- A business process has to be described in a formal language which can be interpreted by every peer. It is also necessary to provide this description to the network, so that every peer which is interested in this workflow has access to the details.
- For a peer to being able to participate in a workflow process, we need to know about its capabilities. It includes a description of the tasks the peer is able to execute in connection with some quality of service attributes like cost or estimated execution time. This information has to be provided to the other peers and serves as a basis for a qualitative election process.
- Within the network we will allow to manage various distinct processes. In order to separate them from each other, it has to be possible to create a community where a specific process is treated and any communication concerning it, takes place. Peers can join and leave communities to shield themselves from unsolicited network traffic.
- To execute a workflow process, one peer may need to act as a coordinator peer. It creates a new instance of the process and has to provide any initial data if applicable. It carries out the peer election for the specific tasks and assigns the peers. Furthermore the coordinator tracks the progress of the overall workflow at any stage and retrieves the output data.

- As all peers work autonomously, they will only be provided with the minimum amount of information necessary. Hence, they have a local view of the problem and don't know about the global scope of the workflow.
- We want to take into account the participation of mobile clients as peers in terms of availability and network load. Such clients will not be permanently connected to the network and will have limited bandwidth compared with other peers.

## Chapter 2

# Used Technology

### 2.1 Peer to Peer

The first peer-to-peer (P2P) technologies emerged more than a decade ago to facilitate communication and resource utilization within the enterprise. Today, P2P describes the general model of using direct communication between all devices on the network. P2P brings connectivity to the edge of the network, enabling any connected device on the network to communicate and collaborate. With P2P, applications can be more collaborative and communication-focused, and information can be more timely and accurate.

While P2P is not a specific architecture or technology, it does enable a number of innovative applications, including:

- Sharing files of all types
- New forms of content distribution and delivery
- Instant messaging and pervasive devices communicating
- Collaborative work and play such as Web-based meetings and interactive gaming
- Distributed search and indexing to enable deep searches of Internet content that quickly yield up-to-the-minute results
- Sharing CPU and storage resources to better utilize capital investments

### 2.1.1 PeerWare

The advantages of a peer-to-peer architecture go well beyond the realm of Internet file sharing, becoming crucial in supporting business processes and especially collaborative work involving mobile users. Targeting on this issue, PeerWare [3, 4] was designed as a core communication middleware for TeamWork applications.

Collaborative work is intrinsically peer-to-peer in nature. Members of a team typically interact directly with each other, with each member being responsible for a given set of documents and carrying with them the subset relevant for discussion. On the other hand, most of the currently available tools supporting collaboration exploit a rigid client-server architecture.

This results in an 'architectural mismatch' between the external view provided by the application and its internal software architecture. The effect of this mismatch is a lack of flexibility in carrying out the interactions, which must all be funneled through the server. This limitation is even more evident when mobility becomes part of the picture. People need to communicate and collaborate even while in movement, and independently of their location. However, in similar situations, server access is often prevented by technical or administrative barriers.

A peer-to-peer approach holds significant advantages over traditional client-server architectures. When a peer-to-peer architecture is adopted, data and services are no longer gathered in a single point of accumulation. Instead, they are spread across all the nodes of the distributed system. Users may directly host the resources they want to share with others, with no need to publish them on a particular server.

Interestingly, these features are relevant not only in mobile scenarios but also in fixed ones, where the decentralized nature of a peer-to-peer architecture naturally encompasses the case of multisite or multicompany projects, whose cooperation infrastructure must span administrative boundaries, and is subject to security concerns.

Unfortunately, most of the peer-to-peer applications developed in recent years started from promises that are rather different from those outlined thus far. They target the Internet and aim at providing peer-to-peer computing over millions of nodes, with file sharing as their main application concern. The difference in perspective from the domain of collaborative work is made evident by their search capabilities, which typically do not guarantee to capture information about all matching files. In most cases they do not take into consideration features like security or the ability to support reactive interactions, which are crucial in cooperative business applications. Moreover, they bring peer-to-peer to an extreme, where the logical network of peers is totally fluid, and no peer can be assumed to be fixed

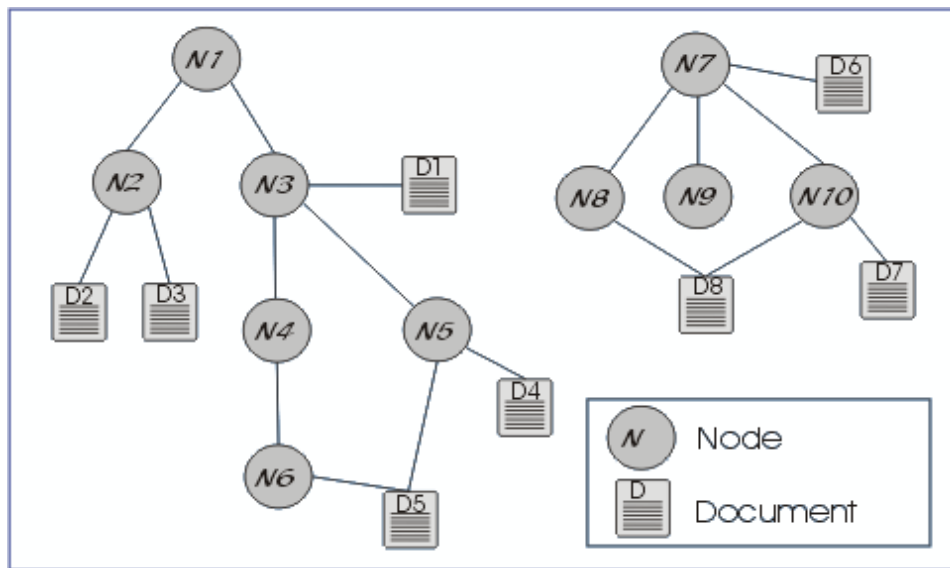


Figure 2.1: The data structure managed by PeerWare.

and contributing to the definition of a permanent infrastructure. This radical view prevents access to resources exported by non-connected peers, which is unacceptable in the business world, where critical data is often required to be always available, independently of its owner.

On the basis of the above considerations, PeerWare was developed: a peer-to-peer middleware for teamwork support specifically geared towards the enterprise domain. PeerWare is both a model and an incarnation of this model in a middleware. In developing both, the first concerns were minimality and flexibility.

**The model** The PeerWare coordination model exploits the notion of a global virtual data structure (GVDS), which is a generalization of the LIME [5] coordination model. Coordination among units is enabled through a data space that is transiently shared and dynamically built out of the data spaces provided by each accessible unit. The data structure managed by PeerWare is a hierarchy of nodes containing documents, where a document may actually be accessible from multiple nodes, as shown in Figure 2.1. This structure resembles a standard file system, where directories play the role of nodes, files are the documents, and Unix-like hard links are allowed only on documents.

When a peer is isolated, it is only given access to its own tree (stored locally) of items (i.e., nodes and documents). However, when connectivity with other peers is established, the peer has access to the virtual tree constructed

by superimposing the trees contributed by all the peers in the system, as illustrated by Figure 2.2.

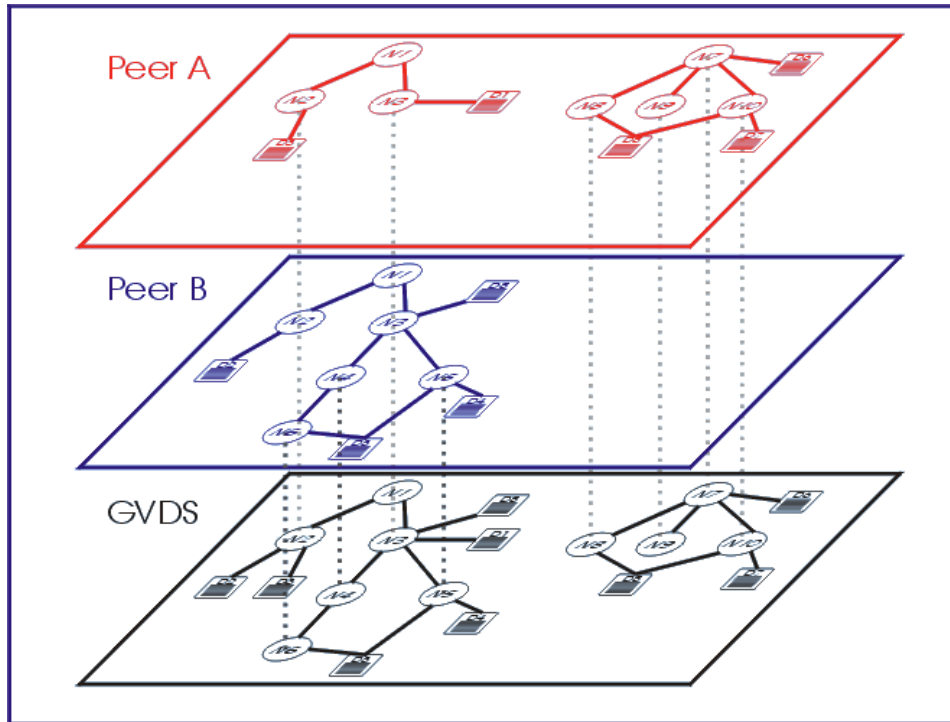


Figure 2.2: Building the GVDS in PeerWare.

In search of minimality, PeerWare provides only three main operations to operate on the GVDS:

- the execute operation allows peers to execute an arbitrary piece of code on a selected set of items held by connected peers. The results are collected and returned to the caller
- the subscribe operation allows peers to subscribe to events occurring on a selected set of items, while
- the publish operation allows peers to notify the occurrence of events.

By exploiting these primitives, peers can query the GVDS and also subscribe to events and receive the corresponding notifications. The hierarchical structure of the GVDS provides a natural scoping mechanism, thus leading to an efficient implementation of searches.

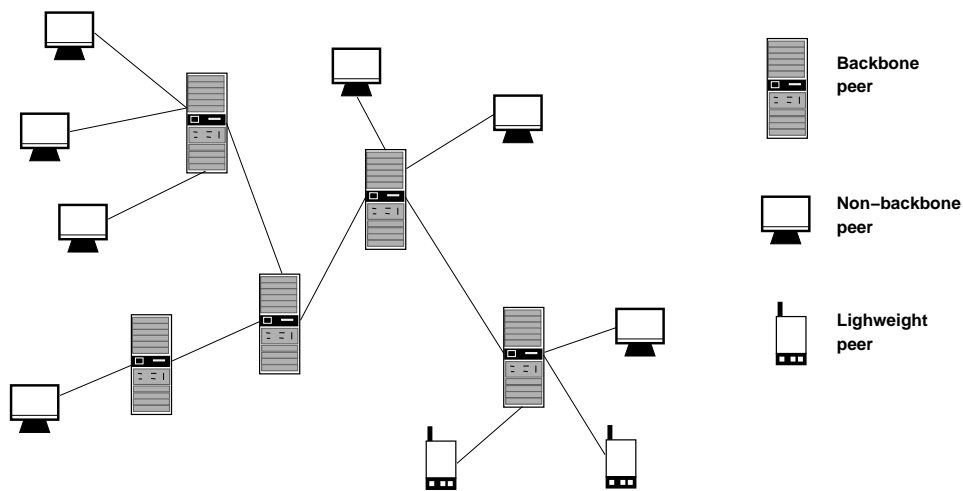


Figure 2.3: The PeerWare runtime architecture.

**The middleware** Currently, the PeerWare model has been implemented in two middleware: one, developed in Java [6], has been used as the core of the MOTION [7, 8] platform, the other, developed in C# [9] under the Microsoft .NET [10] infrastructure, is the core of the PeerVerSy [11] configuration management tool.

Both implementations are tailored to the business domain and distinguish between a set of permanently available backbone peers, and a fringe of mobile peers, which are allowed to connect and disconnect as required. To optimize routing, these peers are connected to form an acyclic graph in which the mobile peers represent the leaves, as shown in Figure 2.3.

Access control and security are critical issues in the target domain and they are addressed by two separate modules. One provides mechanisms to establish encrypted channels among peers and to manage the security information necessary to authenticate a peer. The other embeds the actual security policy that determines the capabilities of a given peer.

To increase flexibility, the functionalities provided by the security modules and also by the repositories holding local documents are sharply decoupled from the specific implementation provided for these functionalities. Thus, the security protocols, as well as the format of the security information used to perform authentication, and the repository effectively used can be changed easily, e.g., to adapt them to the common practice of a specific business environment.

### 2.1.2 MOTION

The MOBILE Teamwork Infrastructure for Organizations Networking (MOTION) project [7, 8] is a highly flexible, open and scalable Information and Communication Technologies architecture for mobile collaboration. It is built on top of PeerWare and provides a range of concepts which can be used in our workflow approach: Users, Communities and Artifacts.

**Sharing data** The notion of community is central in MOTION. The project is actually conceived explicitly to provide support to the cooperative work of virtual community of users. Ideally, when a member connects to the community, he should have the whole knowledge base provided by the community at his fingertips. For instance, a member should be allowed to issue requests for information (e.g., the retrieval of a report matching some keywords or an expert search) without any knowledge about their location. In principle, the member should not even know whether the resource is part of the connected community or not. Similar reasoning applies to the asynchronous forms of communication discussed earlier. In this perspective, the key idea is sharing. No information is required by the initiator of communication, as long as he has the illusion of sharing his own resources with those made available by the community in a shared space of information. For this purpose, some notion of resource space are defined:

- **Resource Space.** It is a set of documents and artifacts. It provides a minimal set of primitives needed to store, retrieve, and query the elements of the set. It can be as simple as an array of objects maintained in RAM, or as complex as an industry-strength DBMS. Each individual owns one resource space. The owner of a resource space has access to all of its content.
- **Member Space.** It is the subset of documents contained in an individual's resource space, and more precisely the set of documents that pertain to a given community. An individual can be member of multiple communities, and she can choose which documents pertain to each community. The same document may be made available to several communities. This means that member spaces can have intersections. A member space can be seen as a view on an individual's resource space.
- **Community Space.** It is the union of all the member spaces belonging to the members of a community. Of course, only the fraction corresponding to the members of the connected community will be directly available.

The notion of sharing that stems from the above definition of community space is indeed a transient one. Since the community space is built only by

the spaces of the members of the connected community, the content of this space will change dynamically according to connectivity. In particular, a given resource will be available to the connected community only as long as its owner belongs to it. Nevertheless, the ability to communicate and share artifacts asynchronously is a key feature of the MOTION platform, and one that requires a notion of sharing that relies on the persistence of resources with respect to the changing community with respect to the number of connected members that compose it. For this reason, a portion of the community space is actually made persistent: we refer to this part of the community space as the community cabinet. Since it is part of the community space, the cabinet is available to all community members. Nevertheless, the content of the cabinet is permanently available rather than transiently built and dynamically changing, i.e., the content is available to community members independently of whether the individual owning a given resource in the cabinet is currently connected.

**Knowledge sharing** Knowledge sharing is a key requirement in MOTION. The system has to provide simple access and manipulation mechanisms for the distributed knowledge base. The knowledge base is distributed over several peers in the MOTION architecture. In MOTION a peer is any computing device that executes the MOTION middleware. In addition, peers can be in disconnected or in ad-hoc mode and therefore not reachable. However, users should be shielded from the complexity of dealing with the actual location of a requested artifact (location transparency). This goal is achieved by introducing the notion of community. A community in MOTION is a set of users, the community members. The members are grouped through some membership relation, e.g., a common interest in the design of the latest cellular phone.

Each user may belong to one or more community. Each user in the system owns artifacts that are stored in the users' resource space. If a user wishes to share a set of artifacts with his colleagues, he makes them available to a community. The subset of artifacts from the resource space that is made available to a given community is called the member space. The member space contains all the artifacts a user wishes to share with a given community. An artifact can also be made available to more than one community. For instance, this paper could be of interest for the community *Software Engineering* and the community *Web Systems*. That is, a user can have more than one member space that may overlap.

The set of artifacts the connected community members contribute to a community is called the community space. The community space is therefore the union of the member spaces of all community members that are

currently connected. From the user view, the access to the artifacts in the community space is transparent regardless of the actual physical location of the artifact. The resource and community spaces have essentially similar basic functionalities that enable the user to query and manipulate the content of the space and to subscribe to occurring events.

The idea of this community space is a dynamic one. Since the community space is built only by the member spaces of the community members that are currently connected, the content of this space will change dynamically according to the users connected to the system. In particular, a given resource will be available to the connected community only as long as its owner is part of it. Nevertheless, the ability to communicate and share artifacts asynchronously is a key requirement for MOTION. This requires that specific artifacts have to be accessible persistently to the users connected to the system. For this, a portion of the community space is actually persistent: we refer to this part of the community space as the community cabinet. Since the community cabinet is part of the community space, the cabinet is always available to all community members. Thus, an artifact stored in the community cabinet is available to the community members regardless whether the owner of the resource is currently connected.

The concept of MOTION communities and cabinets is a flexible concept for knowledge sharing in a distributed environment.

**Web architecture** In this section, we provide an overview of the Web-based MOTION peer-to-peer architecture. Figure 2.4 illustrates typical MOTION peers and their components. It also shows the access to the MOTION system from devices not running a Web-server and the MOTION middleware (Web-terminal, WAP [12] phone, PDA). Each MOTION peer contains both, a Web-server, running a Java servlet engine and a Web-client. The middleware provides an API to connect to the knowledge repository. This API consists of generic functions to manage the artifacts stored in the repository. This architecture enables the support of various kinds of repositories such as XML-databases, SQL-databases, file-systems, etc. Only an adapter between the repository and the MOTION API has to be provided. The repository not only stores the actual artifact, but also XML metadata. The metadata is used for managing and querying the repository. Not every MOTION peer needs to contain the whole machinery depicted in Figure 2.4. Depending on the computational power and the memory capacity, some components might offer reduced functionality. These constraints reduce the functionalities offered to the user. Due to memory limitations, a MOTION peer on a PDA, for example, cannot host a full SQL-database as a

repository. Instead the repository on a PDA could just cache the XML meta-data of the retrieved artifacts and access the actual artifact using the URL. MOTION also supports access from devices not running a Web-server and the MOTION middleware. These devices access the platform via the Web-server of a MOTION peer. The only requirement for these devices is that they run a Web- or WAP-client. This enables the user to access MOTION from any computer running a Web-browser (e.g. in an Internet cafe), from a PDA or WAP-enabled cellular phone.

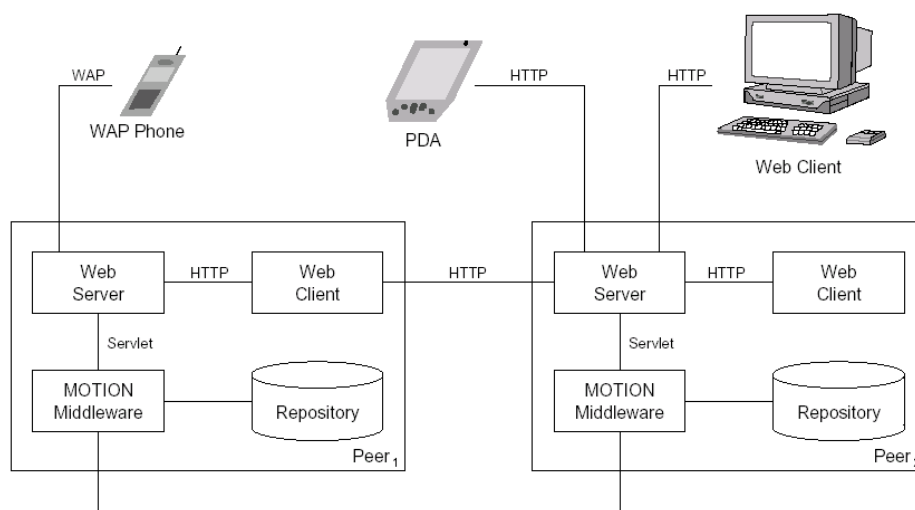


Figure 2.4: MOTION Architectural Sketch

**Repository access** In this section, we show the artifact access to the local repository as well as to the repository on a remote peer. Artifacts that are stored in the local repository are retrieved through the local Web-server, which delegates the request to the underlying MOTION middleware, which in turn queries the local repository. The retrieved artifact is then transferred to the Web-client via the middleware and the Web-server. This scenario is illustrated by the sequence diagram in Figure 2.5.

Artifacts that are not stored in the local repository are accessed through the local Web-server and the MOTION middleware. The middleware is then responsible for locating the artifact among all connected peers. The URL of the requested artifact is provided to the Web-server that forwards

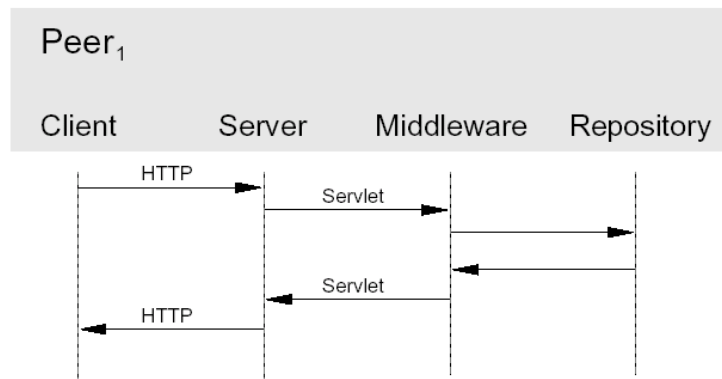


Figure 2.5: Local access to artifacts

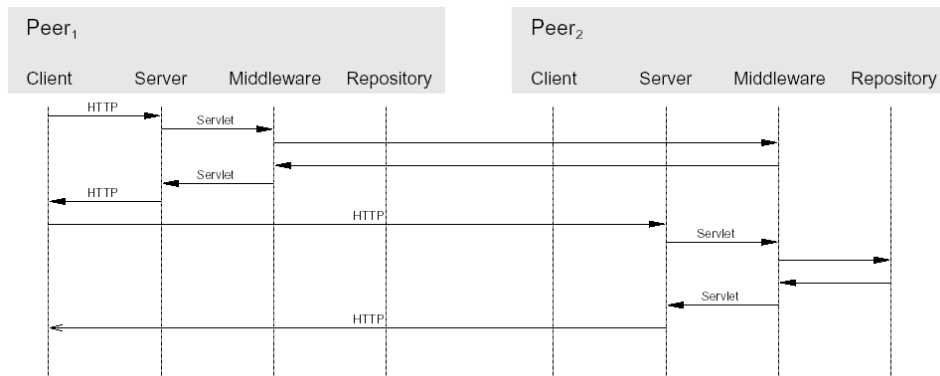


Figure 2.6: Remote access to artifacts

it to the browser. The actual artifact is retrieved via a conventional client/server access using HTTP. In this case, the requesting peer operates as client and the peer hosting the requested artifact as server. Since every peer consists of a Web-server as well as a Web-client every peer can operate in both roles. This scenario is shown by the sequence diagram in Figure 2.6.

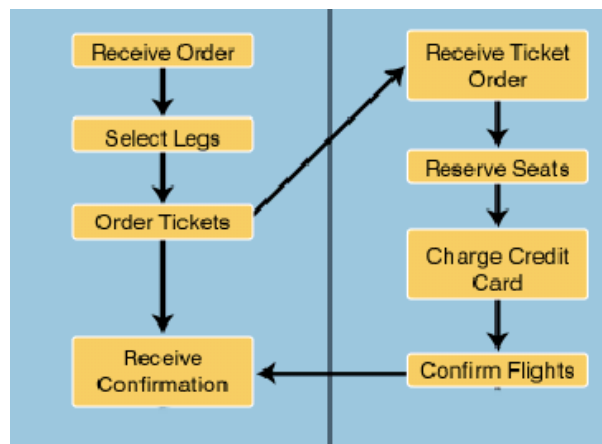


Figure 2.7: Sketch of a workflow scenario

## 2.2 Workflow concept

Due to the increasing complexity of business processes in enterprises, we face the need for computer-aided management systems which help to coordinate the incidental tasks. A workflow management system [13, 14] (WFMS) administrates processes and guarantees the proper execution of all parts. To use a WFMS we have to analyze the business process and set up a formal description. Most problems which occur can be abstracted into several classes. Thus we can create templates for recurring processes and apply them on a specific environment. Figure 2.7 shows a sketch of a typical workflow.

Activities are depicted with boxes with the name of the activity inside. The arcs between the boxes describe the dependencies. This leads to a well defined chronological execution of the tasks. In this example the workflow starts with the activity *Receive order*. The next task to be performed is *Select legs* which can be started after the previous task has been completed. In a workflow execution there are also documents involved. Typically we have an input document and an output document. In this example this would correspond to a booking order with any information required and a booking confirmation. Furthermore an activity can produce information which is required by another dependent activity to perform it's work. This information is hidden to the outside. Any document which occurs in a flow is required to satisfy certain criteria, both in terms of structure and completeness.

## 2.3 Description Language

As shown in the previous section, we need a formal language which allows us to model a business process. It must satisfy the following requirements:

- We need to distribute the model to certain participants. Therefore it has to be simple to distribute, e.g. in terms of size.
- A business process consists of several tasks which the language has to be able to represent.
- It is necessary to provide ways to define dependencies between task (control flow).
- Input and output documents have to be specified if applicable (data flow).
- Some tasks will produce data and others may consume it. Any documents which the tasks take as an input or output parameter have to be stated to ensure the document flow.
- We want to use an existing and standardized language.

### 2.3.1 Business Process Execution Language for Web Services

The Business Process Execution Language for Web Services (BPEL4WS) [31, 32] provides an XML notation and semantics for specifying business process behavior based on Web services. A BPEL4WS process is defined in terms of its interactions with partners. A partner may provide services to the process, require services from the process, or participate in a two-way interaction with the process. Thus BPEL orchestrates Web services by specifying the order in which it is meaningful to call a collection of services, and assigns responsibilities for each of the services to partners. It can be used to specify both the public interfaces for the partners and the description of the executable process.

Business processes can be described in two ways. Executable business processes model actual behavior of a participant in a business interaction. Business protocols, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes. BPEL4WS is meant to be used to model the behavior of both executable and abstract processes. In our approach to model workflow processes we focus on abstract processes only.

BPEL4WS is layered on top of several XML specifications: WSDL 1.1 [33], XML Schema 1.0 [34, 35, 36], and XPath 1.0 [37]. WSDL messages and XML Schema type definitions provide the data model used by BPEL4WS processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services. BPEL4WS provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

## 2.4 XML Data Manipulation

A main issue in our approach is to minimize traffic and to deal with participants which are not permanently connected. To achieve this, we have to store some relevant information of other peers in a local database persistently. In view of mobile clients and their limited resources, we will not be able to use an ordinary RDBMS (Relational DataBase Management System). A slim XML database is highly suitable for our needs, as implemented in the PDOM Component and the XQL Engine [38]:

### 2.4.1 The PDOM Component

The PDOM (Persistent Document Object Model) component stores XML documents in a compact, binary format. The PDOM object manager translates method calls to the standardized W3C-DOM API [39] into operations on the binary files. This is done transparently for the application layer. Thus read and write access to arbitrarily large XML documents become possible, without the necessity of any modification to application programs. Even without caching the PDOM has a throughput of more than 3 MB of XML data per second. A scalable, self-optimizing cache can further improve performance. All methods of the PDOM are implemented in a transaction-safe way - even in multi user environments. In addition explicit commit points can be set. For many scenarios the PDOM offers a lightweight and often more efficient alternative to complex and less flexible DBMS based solutions. As the PDOM can deal with well-formed XML, in particular the costly spadework of schema design is avoided.

### 2.4.2 The XQL Processor

The Extensible Query Language (XQL) is a declarative, path-oriented query language for XML. It includes most operations familiar from SQL [40], e.g. selection, restructuring, joins, and views. However, XQL considers the semi-structured nature of XML. Introduced first at W3C's [16] conference on XML query languages, XQL has since been implemented by several namable IT-vendors. The XQL processor implements the full XQL

proposal together with some extensions. These include parallel retrieval of distributed documents, automatic translation of HTML to XHTML [17], and user defined extension functions. The implementation realizes a robust and efficient mix of algebraic and physical query optimization techniques. This results in leading edge performance fully competitive with commercial XML data servers. The processor can be used on top of any W3C compliant DOM implementation, especially the PDOM. Any data source can be made queryable by the XQL processor by implementing a wrapper which maps the source's behavior and data onto the DOM API. This allows for seamless and simple embedding of XQL support even into existing applications and services.

## 2.5 Related Work

### 2.5.1 JXTA

Until recently, however, P2P technologies have been used primarily in single-function applications, such as instant messaging. Taking the concept of P2P much farther, Sun Microsystems [19] founder and chief scientist Dr. Bill Joy conceived the idea of JXTA [18] technology as a means of integrating P2P into the very core of the network architecture. JXTA technology is a set of simple, open peer-to-peer protocols that enable any device on the network to communicate, collaborate, and share resources. JXTA peers create a virtual, ad hoc [20] network on top of existing networks, hiding their underlying complexity (see figure 2.8). In the JXTA virtual network, any peer can interact with other peers, regardless of location, type of device, or operating environment - even when some peers and resources are located behind firewalls or use different network transport protocols. Thus, access to the resources of the network is not limited by platform incompatibilities or the constraints of a hierarchical client-server architecture. JXTA technology espouses the core technology objectives of ubiquity, platform independence, interoperability, and security. JXTA technology runs on any device, including cell phones, PDAs, two-way pagers, electronic sensors, desktop computers, and servers. Based on proven technologies and standards such as HTTP, TCP/IP and XML, JXTA technology is not dependent on any particular programming language, networking platform, or system platform and can work with any combination of these.

Interoperability is a central goal, and JXTA technology is designed to enable interconnected peers to easily locate and communicate with each other, participate in community-based activities, and offer services to each other seamlessly across different platforms and networks. Integrated security mechanisms such as Transport Layer Security (TLS) [21], digital certificates, and certificate authorities help ensure security while facilitating free-flowing communication.

Both PeerWare and JXTA are device independent middlewares which provide basic functions necessary to support peer-to-peer applications. They build and give access to a virtual dataspace containing all resources of the connected peers. As the both technologies are very similar, JXTA could be a good alternative for PeerWare as the basis of the MOTION middleware.

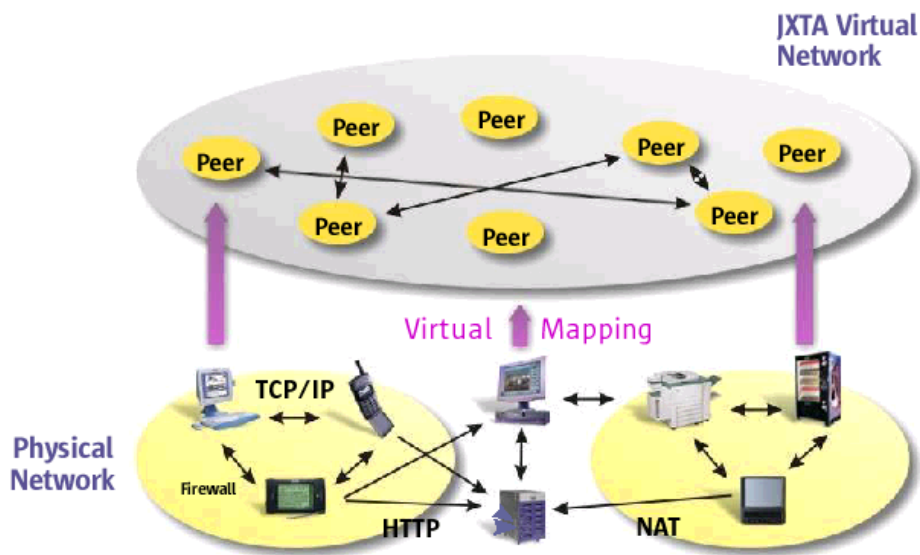


Figure 2.8: JXTA Virtual Network.

### 2.5.2 Web Workflow Peers

The peer-to-peer architecture introduced in [22] is based on the concepts of the Web Workflow Peer (WWP) and the Web Workflow Peer Directory (WWPD). A WWPD is a directory system which provides a list of all peers (WWPs) available to participate in Web workflow processes. Peers can register at this directory and offer its services and resources to other peers. The WWPD also assists WWPs to locate other peers and use their resources. The architecture is completely decentralized, as no central workflow engine is used to coordinate the process execution. The server functionality and data are distributed among the WWPs. The administration is achieved using a notification mechanism. For instance, at the completion of an activity, the WWP notifies the Administrator Peer so that he can update the status of the process instance.

**The WWP Directory (WWPD)** The only centralized feature in this system is the WWPD. The concept of centralization can also be found in various other peer-to-peer architectures, e.g. KaZaA [23], SBARC [24], or Gnutella [25, 26]. Peers can register with the WWPD and advertise the services they provide. Similar to the business services of UDDI [27] it manages a list of WWP profiles which include IP address, list of provided tasks and administration data. As WWPD is an active directory, it maintains information about peer availability (e.g. by 'pinging' their IP addresses) and quality aspects as connection speed and packet loss.

**The Web Workflow Peer (WWP)** A Web Workflow Peer is a processing capacity which can be accessed using Internet protocols, similarly to a web service. A WWP that initiates and administers the process is called the Administrator Peer. To do so, the Administrator may have to reallocate (re-assign) or cancel activities or issue deadline alert notifications to Participating Peers. Other WWPs delegated to carry out workflow activities are called the Participating Peers. The functionality of the Participating Peer is to receive activities and to receive/send notifications. It is free to delegate work to other peers. Although conceptually there is a difference between these peers, in practice all peers are capable of acting as both roles.

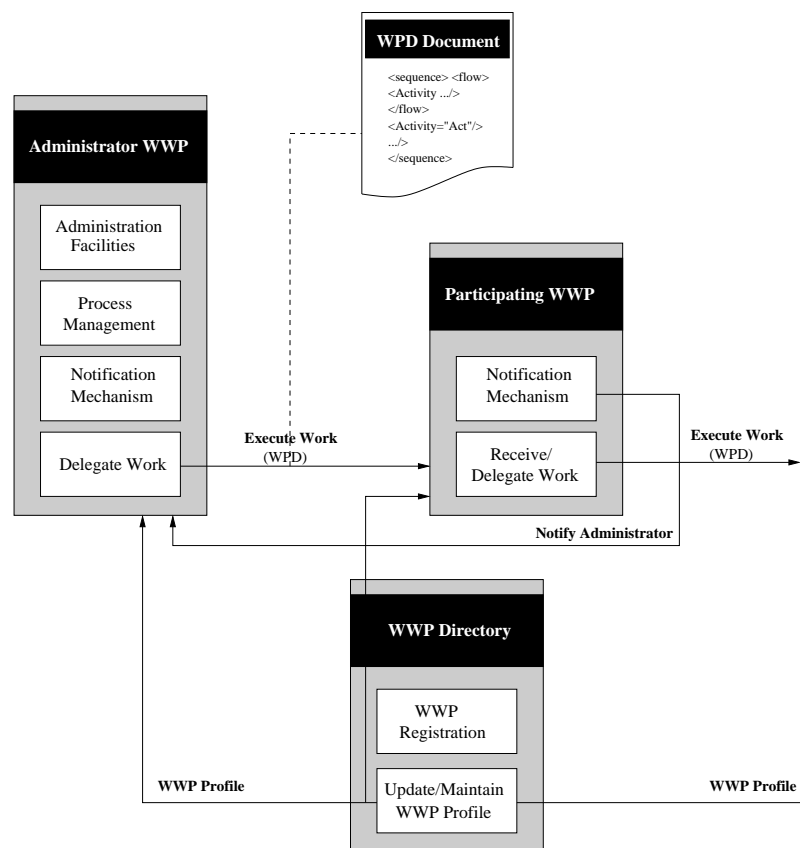


Figure 2.9: The WWP architecture

**The Workflow Process Description (WPD)** The Workflow Process Description is an XML document containing the data and meta-data of a process instance. This is the structural information about the process and links (URIs) to resources, e.g. documents. The WPD document is transmitted from peer to peer attached as a parameter to a message. On execution, the

peers update the Workflow Process Description and decide which WWP needs to be activated next in the process chain.

**Workflow administration and peer notification** The Administrator Peer needs to know the state of each activity at any time. For this reason, WWPs are able to notify others when their activity is completed. Notification messages are structured in XML documents. The system supports different notification messages which inform about completion, rejection, cancellation, and reallocation of activities.

The target domain of the Web Workflow Peer concept is similar to our approach. Both model human workflow scenarios which are defined in structured XML documents. The executing peers are dynamically elected using quality criteria and the task assignment is performed on the basis of negotiations between peers, though there are conceptual differences in centralization issues. In the WWP approach, the WWP Directory is a centralized service which manages task announcements and the peer assignment is performed by the Participating Peers. In our workflow system, the peers manage task announcements decentralized and the coordinator is responsible for peer assignment.

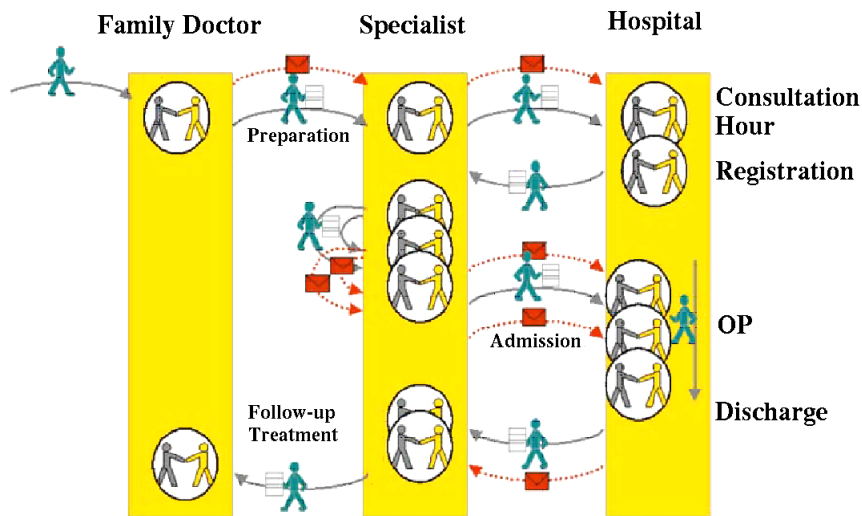


Figure 2.10: Healthcare service example

### 2.5.3 Serviceflow

The serviceflow [28] approach was designed to manage inter-organizational service processes. It aims to support meta-services which are defined as a number of subsequently executed services, offered and carried out by different organizations. Taking a typical example from the healthcare domain, a patient goes through various steps in case of a surgical operation (figure 2.10): He typically starts with consulting a family doctor, is directed to a specialist, chooses a hospital, goes through consultation and registration at the hospital with a schedule for further preparation, passes through all stages of preparation, stays in the hospital where the operation is performed, all of which is followed by aftercare treatment at specialists.

#### The serviceflow concept

The serviceflow approach exploits process-related and Internet technology, with additional focus on customer-related aspects. It is grounded in a twofold perspective on service processes: considering the relationship of services as well as the necessity of their efficient performance. In particular, a serviceflow is defined in terms of service points (figure 2.11). Each service point captures specific service tasks to be carried out and their respective pre- and postconditions from the provider's point of view. The pre- and postconditions represent the contract for interrelating the service points. Service tasks are modeled as UML use cases with each use case being further linked to a rich description, a scenario, and a use case picture

(see [29]). Cooperation pictures can be included the serviceflow representation to further illustrate cooperation among the actors involved.

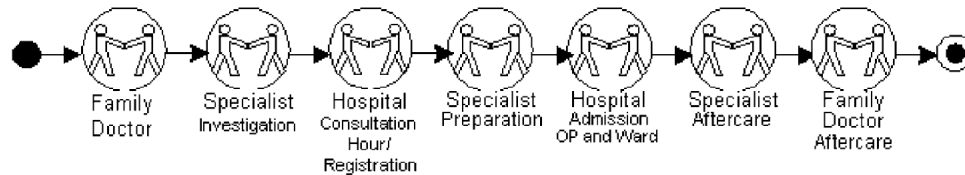


Figure 2.11: Serviceflow model for the healthcare service example

The overall concept for serviceflow management is centered around the technical representations of the modeled process patterns that lead to the notion of service float and service point script. Service floats are sent from service point to service point and capture personalized, always up-to-date process knowledge, whereas service point scripts support and document the standard and adaptable activities at each service point.

Flow models are understood as standard pattern during both, design and execution. Moreover, during execution they become accessible and alterable by service workers while still being interpretable by executing engines. This kind of support bears the following potentials:

1. Initializing a service float by copying and adapting a standard serviceflow pattern guides the provider as to how to deliver the service
2. Enabling providers to access and update the process representations allows for flexibility and instant realization of changes.
3. The update of the current and setting of the next service points forms a basis for automating the delivery of service floats to the next provider.

### Realization

The serviceflow management approach consists of two functional components: The exchange of service floats is accomplished by the *service float application* at each site, which apart from the routing support offers methods to update and query a service float. Furthermore, it includes a storage component where the service float masters are stored (at the starting node in the network). The *service point application* receives and delivers messages, compares messages with preconditions of service points to inform workers about a changed status, and serves to integrate the service point tasks with applications available in the organizations, such as databases.

The decentralized architecture serves processes with "chained execution" [30], where each of the tasks at one providers site is completed before the next provider is in charge, i.e. where no parallel execution is required. To achieve a parallel execution of serviceflows, the system can be configured to work partially or even fully centralized. In a partially centralized architecture, an additional server for documents is added, with the documents no longer being part of the service floats. In this case they will be shareable among different serviceflows and updateable after a service point has ceased to be active. A further step is made if all architectural serviceflow components are united at one central server, managed by an application service provider in charge.

In contrast to our approach, the serviceflow concept models static service processes, as the participating peers are predefined and no peer assignments take place. In the decentralized architecture the tasks are subject to chained execution, whereas in the partly or fully centralized architecture the tasks can be executed in parallel like in our workflow system. To exchange documents, both approaches use URIs to locate the peer the data can be obtained from. Although both the serviceflow concept and our workflow system manage human workflow models, they focus on different domains due to flexibility issues.

## Chapter 3

# Conceptual Design

### 3.1 The scenario

To bring the example of the first section into a practical context, we introduce an environment, in which the execution will be demonstrated in several parts of this thesis. In order to keep the scenario practical and comprehensive, we will minimize the number of peers involved. A suitable configuration is:

- A *Coordinator Peer* which will do any administrative work prior to the execution of the process.
- 3 *Worker Peers* which represent a Traveler, an Agent and an Airline respectively.

In the next sections all steps the peers have to perform in order to have the process instance executed will be explained.

## 3.2 Software Architecture

To implement a peer-to-peer workflow system we have chosen to use the MOTION middleware which is based on PeerWare. It provides us with some useful concepts which will be discussed later.

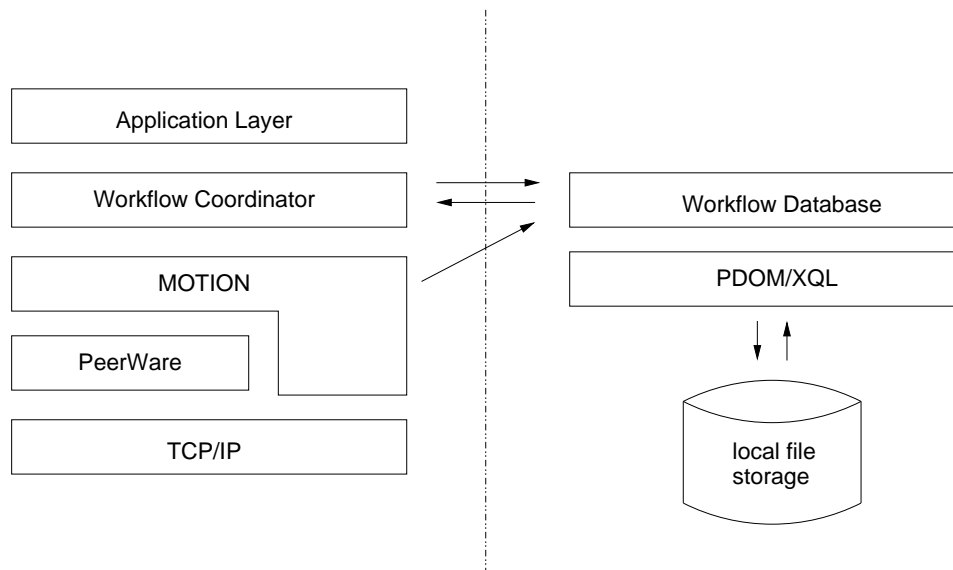


Figure 3.1: Software Architecture

Our approach consists of two major components: The *Workflow Coordinator* and the *Workflow Database*.

### 3.2.1 Workflow Coordinator

It is built on top of the MOTION middleware and is the central part of the peer-to-peer workflow engine. The Workflow Coordinator provides most of the functionality which is required by an application layer, like

- User Management
- Community Management
- Process/Instance Management
- Communication between peers

It is tightly coupled to the Workflow Database which holds information of all known peers. To ensure that the peers are loosely coupled, it retrieves all data from the database, if available, and avoids to actively query other peers if not necessary.

### 3.2.2 Workflow Database

It is responsible for gathering of information from other peers and serves the Workflow Coordinator for storage and retrieval of data involved in the workflow engine. The Workflow Database listens to the messages received by MOTION, analyzes them and saves the relevant data to the storage. It doesn't actively send messages to other peers.

We can distinguish two separate parts within the database:

**Process data** holds detailed description of all known processes. This includes the structure of involved tasks and input/output data.

**User data** is a collection of all known users. For each user it is stored the tasks it has announced and process instances it takes part with detailed status.

The database is persistently stored in the local repository and is accessed using PDOM and XQL technology.

### 3.3 Communities

A main concept in the MOTION middleware is the notion of community. It is a collection of several peers (community members) which semantically share the same interests. A peer can be member of several communities. This leads to the fact, that communities can overlap.

We use this concept at three different levels, which are illustrated in figure 3.2:

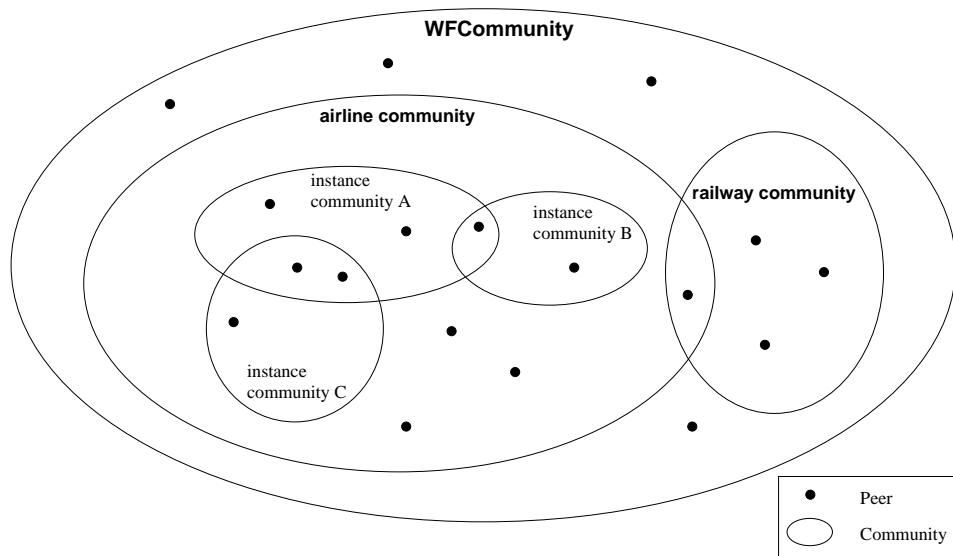


Figure 3.2: Community structure

**WFCommunity, a meta community.** All participants of the workflow system have to be member of the top-level community WFCommunity. It serves as the basic communication platform. Messages of global interest are announced here. It is created during installation of the workflow system and can't be modified or deleted.

**Process communities.** We also use communities to group peers with similar interests. Taking into account our example, it is suitable to create a community for airline reservation issues. Any peer which wants to take part in the execution of an airline process has to become member of this community. Any communication concerning this process is carried out within the airline community. This concept shields peers which are not members of this interest group from receiving unsolicited messages. Process communities can be created by any peer who wants to set up a

separate environment for execution of processes.

In our example the Coordinator Peer has to set up a community for airline reservation issues and will name it *AirlineCommunity*. The three peers acting as traveler, agent, and airline join this community in order to receive relevant messages for this topic. The steps included can be seen in figure 3.3.

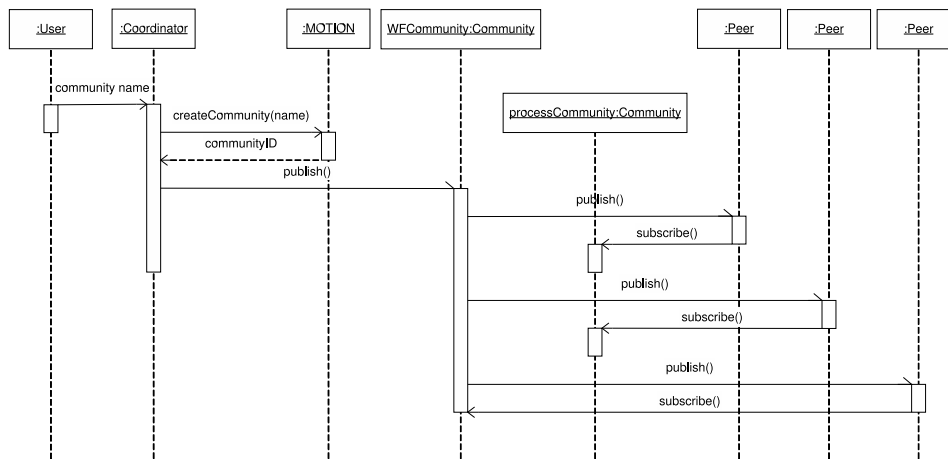


Figure 3.3: Create and join a community

**Instance communities.** Upon creation of a process instance an *instance community* will be established automatically. The peers taking part in the instance execution will automatically become members of the instance community. It serves as the communication platform for any message exchange and information sharing concerning the instance. This concept helps to further reduce traffic within the peer-to-peer network. As illustrated in figure 3.2, instance communities are subsets of process communities which are part of the top-level community WFCCommunity.

## 3.4 Processes, Tasks, and Instances

### 3.4.1 Process

As discussed earlier, we want to model business processes. The formal description of its structure is specified using a BPEL document. It defines the tasks involved, the relationship between the tasks, parameters and data flow. In order to work with this model, it has to be imported into our workflow system by a peer. Once this is done, it is available to all peers and is referred to as a *process*. To import a new process, the peer has to choose a BPEL document containing the description and must select a community which the process will be published in. The Workflow Coordinator stores the information into the Workflow Database and creates a MOTION artifact for the BPEL file. The artifact is then added to the given community and can be downloaded by other peers. An illustration of these steps can be seen in figure 3.4.

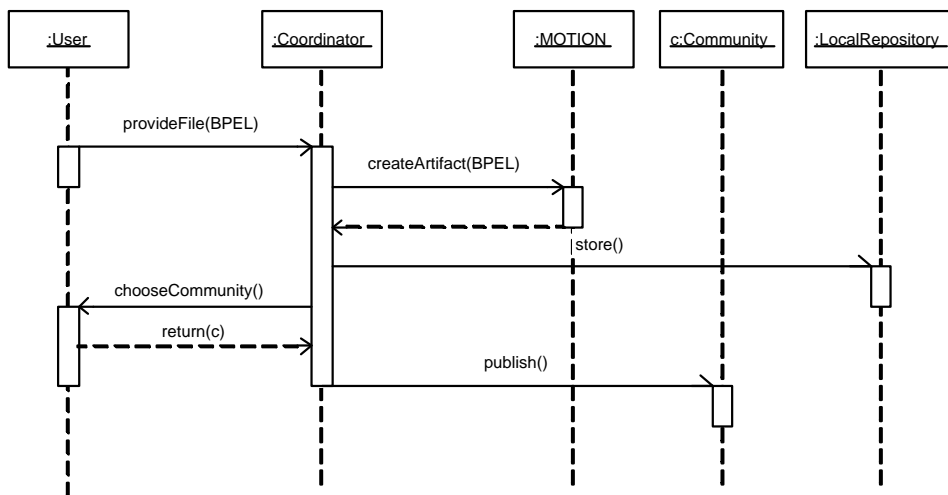


Figure 3.4: Import a new process

To publish the *AirlineReservation* process, the coordinator will add the process description to the *AirlineCommunity*. From this time, the members of this community have access to the description and can download it to their local repository on demand which is illustrated in figure 3.5.

### 3.4.2 Task

In order to distribute pieces of a whole process to several peers, we need to divide the process into several parts. We call these parts *tasks*. The process description gives us a detailed definition of the tasks which includes:

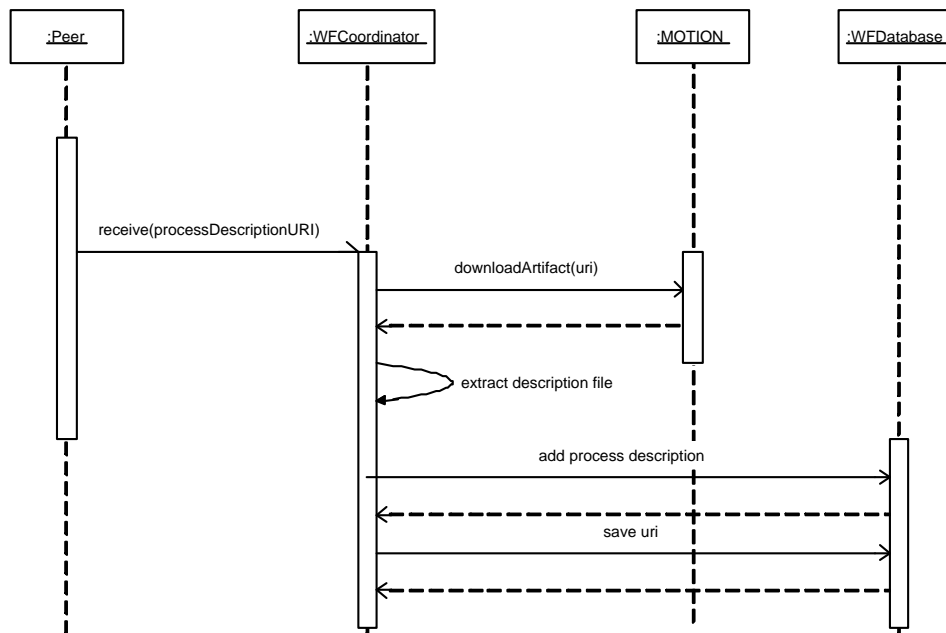


Figure 3.5: Download process description file

- Dependencies between the tasks. It shows the sequence which the tasks have to be executed in.
- Required input data. This can be data which is provided by the Coordinator Peer upon creation of the instance or output data produced by another task.
- Produced output data. It can either be required by another task as input or it can be final output data by the process.

The goal of our workflow engine is to distribute all parts of a process to other peers and manage its execution. If a peer wants to perform a task of a process, it has to publish this to the particular process community (figure 3.6). This also includes giving quality of service attributes which will help the Coordinator Peer to elect a suitable partner. In order to know the task details, the worker peer has to download the process description file from the community if it is not available in its local database.

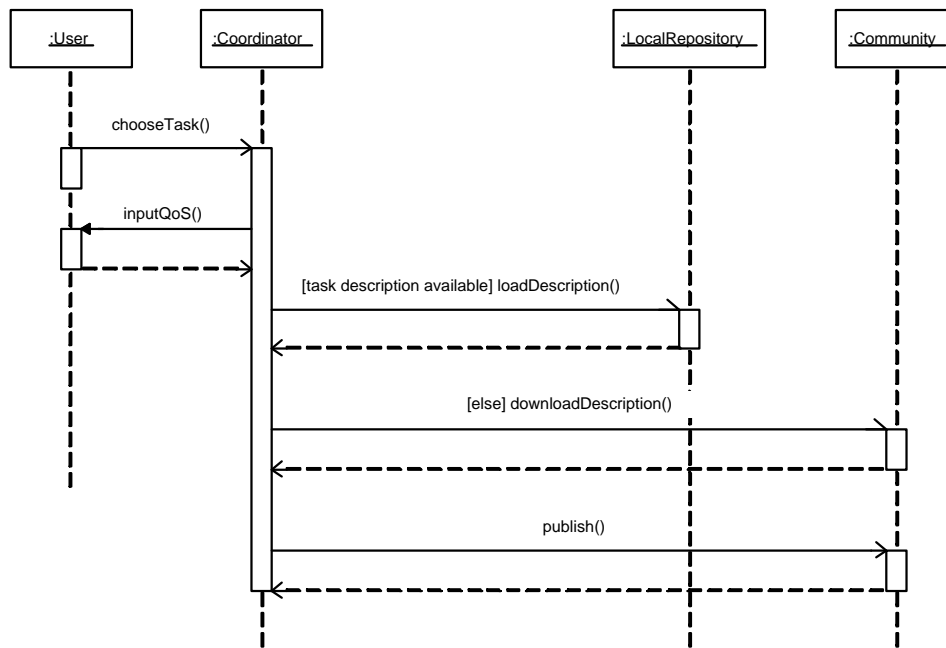


Figure 3.6: Provide a task

If we take a look at our example, the airline reservation process is divided into several tasks. In a scenario with three peers acting as a traveler, agent, and airline respectively we will have a partition of the tasks as shown in figure 3.7. In order to participate in an instance execution, they will have to provide each of the tasks they are supposed to do. Before this can take place, the peers have to ensure that they have downloaded the process description file to their local database.

Traveler	Agent	Airline
Plan Trip	Receive Order	Receive Ticket
Submit Order	Select Legs	Reserve Seats
Receive Itinerary	Order Tickets	Charge Credit Card
Receive eTicket	Receive Confirmation	Confirm Flights
	Generate Itinerary	Issue eTicket
	Issue Itinerary	

Figure 3.7: Task partition of the airline reservation example

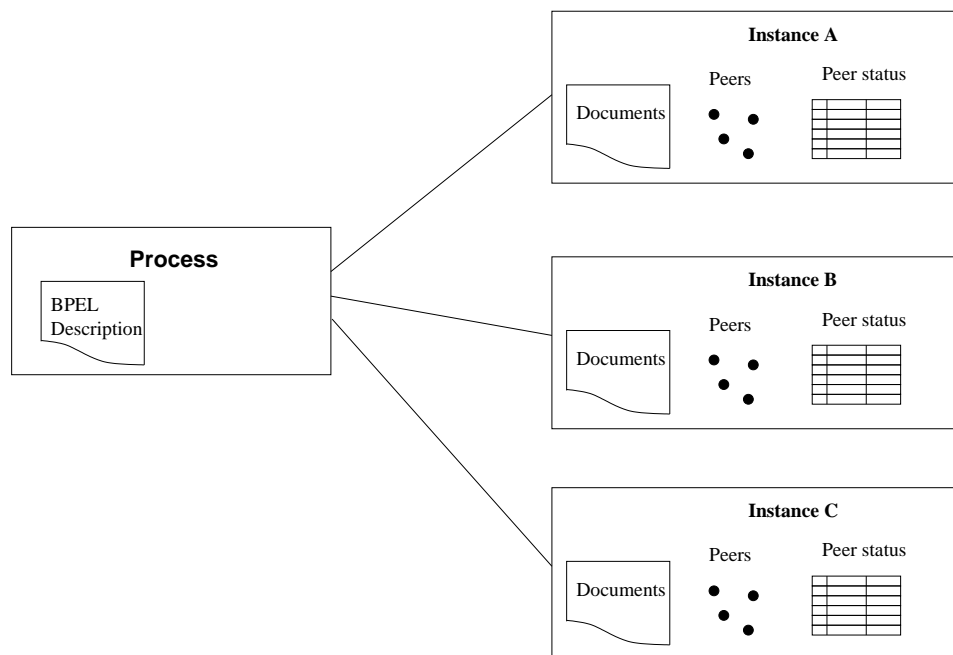


Figure 3.8: Process instances

### 3.4.3 Instance

To execute a process, a peer can create an *instance* of it. The peer becomes coordinator of this instance and is responsible for task assignment and providing input data if applicable. The instance contains any data concerning the execution, like

- assigned peers and details about their tasks
- workflow status of the tasks
- documents involved.

This is illustrated in figure 3.8. A process can be instantiated more than once. To ensure a separate environment for each instance, the peers which participate in the execution are grouped together into a private community, an Instance Community.

To create an instance the coordinator has to choose a process which it wants to execute (figure 3.9). The Workflow Coordinator creates an instance community for this instance in which the relevant information will be exchanged. The new instance will then be published to the process community.

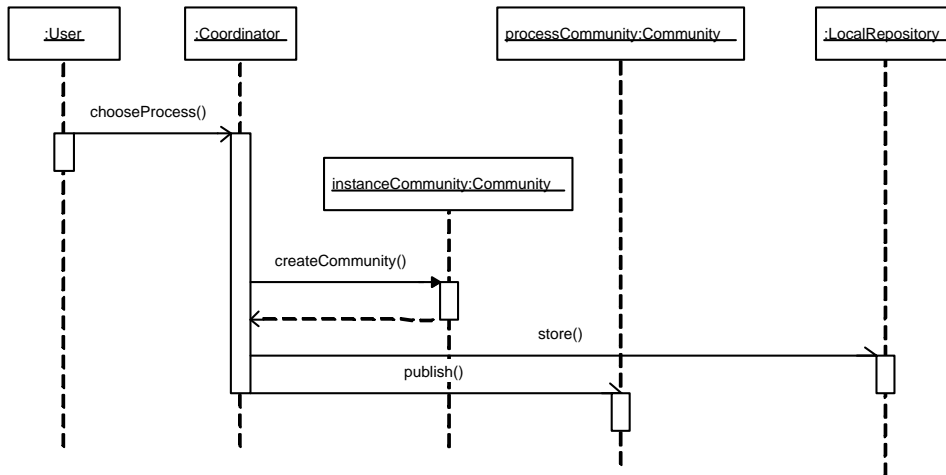


Figure 3.9: Create an instances

To execute our *AirlineReservation* example, the coordinator chooses the process which has already been added to the corresponding process community. An instance community is being set up and the required environment is established.

### 3.5 Quality of Service

In an environment with a large number of peers, we will have a few ones acting as Coordinator Peers and carrying out process executions and the vast majority offering and performing tasks for them. When the Coordinator Peer has to carry out the election process, it is likely that he can choose between several providers. To be able to make the best decision which peer to assign, the peers have to give quality of service (QoS) attributes for each task they publish to the community. The attributes are included in the task announcement which is sent to the community and therefore available to all members. Quality of service attributes can be *time to execute* or *price* for a task. These attributes don't have any influence on the quality of the output data a task may have to generate.

## 3.6 Data distribution

The distribution of information in our workflow system is subject to ensure low bandwidth consumption and to avoid sending unnecessary messages. The concept of community which is outlined in the previous chapter helps to ensure these requirements. The communication between peers is based on message delivery and the exchange of data files wrapped into MOTION artifacts.

### 3.6.1 Messages

Messages are directed notifications from a peer to another peer or a community, implemented as MOTION Messages. They can be targeted to a community referring to the group of peers which most probably are interested in this information or it can be sent to a specific peer. Messages have the function to control the workflow process and to actively distribute information to the peers. Depending on the scope of the message we can distinguish the following types:

**Global Messages.** These messages are global announcements which are sent to the top level community WFCcommunity. We have two message types in this section:

- **Create/Remove a Process Community.** If a peer decides to create a process community, this has to be made public to all members of the workflow system. The peers then can decide whether they are interested in this field and join the community or they don't.
- **Global search for Task announcements.** Peers which are not permanently connected to the network have incomplete knowledge of task announcements. In offline mode they can't receive messages and are not informed about other peers' tasks. A global search queries all workflow members whether they offer a specific task and thus gets an actual view of online peers.

**Process Community Messages.** This message type is sent to process communities and informs the peers subscribed to this community. We have one message here:

- **Provide/Revoke a task announcement.** If a peer decides to newly provide a task, to change its quality of service attributes, or to stop offering a task, it informs the community. The members update their local database to keep track of the peers capabilities.

**Direct Peer to Peer Messages.** As the name says, these messages are sent directly from one peer to another. They are used for communication between a coordinator and a peer which is involved in the coordinators instance.

- Negotiation of peer assignments. If a coordinator wants a peer to execute a task, it has to ask the potential partner if it is willing to participate in this instance. This communication is relevant only for these two peers and the messages can be exchanged directly between them.
- Update of a tasks status. A peer which is assigned to execute a task has to inform the coordinator of the status of its work. If a new status is reached the peer sends a notification message to the coordinator.

### 3.6.2 Data Files

To distribute data files between peers we make use of MOTION Artifacts. An artifact is a wrapper for data files which can be published to communities. The members of that community can download the artifact if they require it. The workflow engine has to distribute data files in two different cases:

**Process description files.** We have to ensure that all peers have access to process descriptions, which are provided as BPEL files. A coordinator which introduces a new process has to publish the corresponding BPEL file to the community he wants the process to execute in. Potential participants of this process can download the artifact into its repository and thus retrieve the process description.

**Instance data files.** We have to provide a mechanism which allows peers to exchange data involved in a workflow execution. According to the process description, peers which execute a task in a process instance may have to produce output data to fulfill their duty. To do so, they wrap the corresponding files into artifacts and publish them to the community. Peers which perform dependent tasks can then download the required artifacts.

### 3.7 Instance lifecycle

As we have seen, process instances are environments for executing processes. One is created when a coordinator decides to carry out a process and is removed from the workflow system when it is finished and the output data is collected. In general we have consecutive five steps in the life of an instance (figure 3.10):

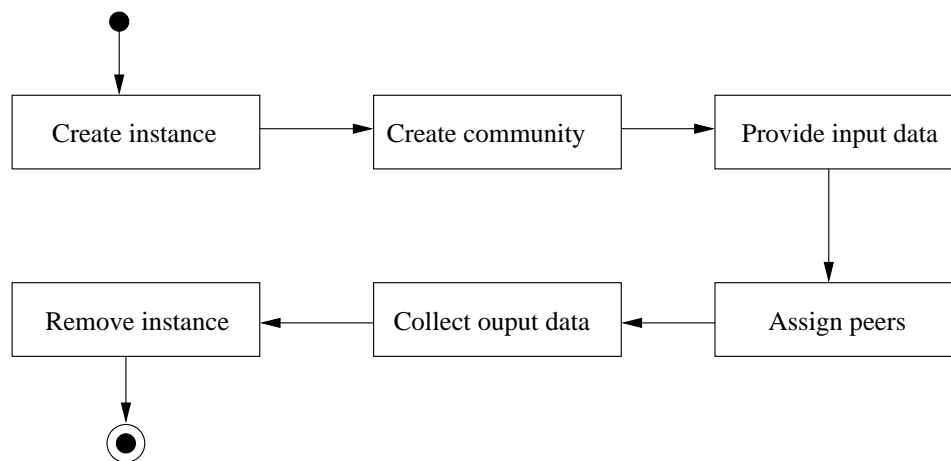


Figure 3.10: Instance lifecycle

1. **Create instance.** A coordinator invokes the creation by specifying the process he wants to be performed.
2. **Create community.** Automatically an instance community is created which encapsulates the instances messaging and environment.
3. **Provide input data.** If any data is required for the execution, it has to be supplied before the tasks can start.
4. **Assign peers.** For each task a peer has to be assigned which is responsible for performing the specific part of the process.
5. **Collect output data.** The coordinator collects the output data specified by the process description, which is the result of the overall work.
6. **Remove instance.** Having successfully received the output documents, the instance and its community are automatically removed from the workflow system.

### 3.8 Task status

A task which is published by a Worker Peer can be requested by any Coordinator Peer. This starts a negotiation between them and will lead to the execution of the task, if both of them agree to work together. From the coordinator's request to the delivery of the final result by the worker, the task goes through several stages. Each stage describes the status the task currently is in. A task status is always bound to a Coordinator peer. This means, that a task which is consumed by different Coordinator Peers will have a separate status for each of them. In Figure 3.11 we have an illustration of this behavior. Consider that a specific peer provides several tasks, including *reserveSeats* and *bookFlight*. The figure shows a scenario, where two different coordinators currently have negotiations: The first one is already consuming the task *bookFlight* and has requested the task *reserveSeats*. The second coordinator has requested *bookFlight*.

#### Peer 1

Tasks	Status	
	Coordinator 1	Coordinator 2
reserveSeat	REQUESTED	—
bookFlight	STARTED	REQUESTED
.....	.....	.....

Figure 3.11: Table of peer status

The sequence of all possible status are shown in figure 3.12. The transitions occur in the following cases:

1. If a Coordinator Peer wants to consume a task, he has to send a request to the peer offering it. It is marked as *REQUESTED*.
- 2a. The Worker accepts to perform the task the appropriate message. The status now is *ACCEPTED*.
- 2b. The Worker is not interested in doing the job and returns a refusal. The task is marked as *REFUSED* and the negotiation is finished.
- 3a. The Coordinator assigns the Worker and sets the task to *ASSIGNED*.
- 3b. It is also possible, that the Coordinator now refuses to assign the task to a specific peer. This can happen e.g. if he was able to find another

one to carry out this part of the process. He informs the Worker of his decision and the status turns to *REFUSED*.

4. When the Worker has all required input documents and doesn't depend on another peer any more, it immediately starts work and marks the task as *STARTED*.
5. Upon finishing the work and delivering any output documents, we have reached the final status *FINISHED*.

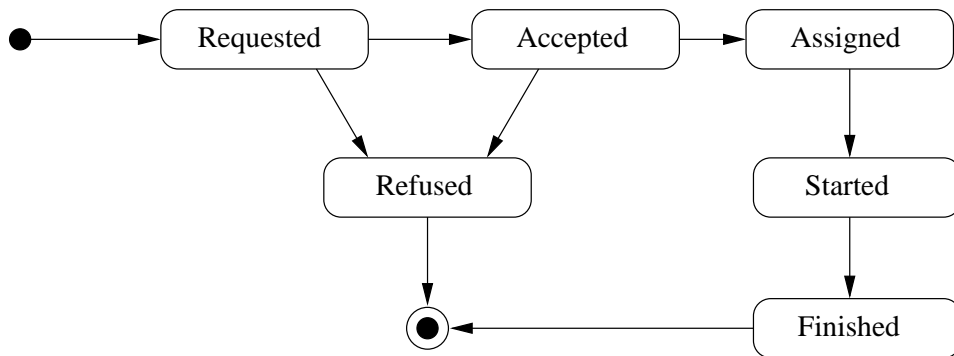


Figure 3.12: Task status

## Chapter 4

# Use cases

The functionality of our workflow system can be divided into three main parts. The *Process Management* basically contains methods for handling process descriptions and communities (figure 4.2). The *Task Negotiation* deals with the possible actions from requesting a task to assigning a task (figure 4.3). The *Instance Execution* part has the functionality of starting and finishing tasks and instances and management of data flow (figure 4.4).

In our use cases we can distinguish two actors. The *Worker Peer* acts as a service provider who offers tasks and performs them on demand. The *Coordinator Peer* manages processes, creates instances, and elects Worker Peers which have to execute tasks autonomously.

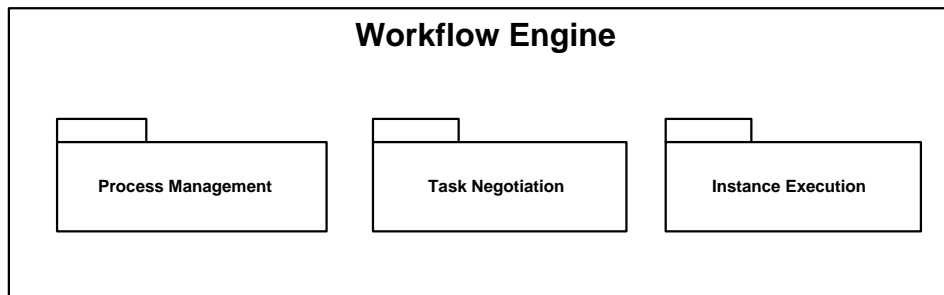


Figure 4.1: Use case: Workflow System

## 4.1 Process Management

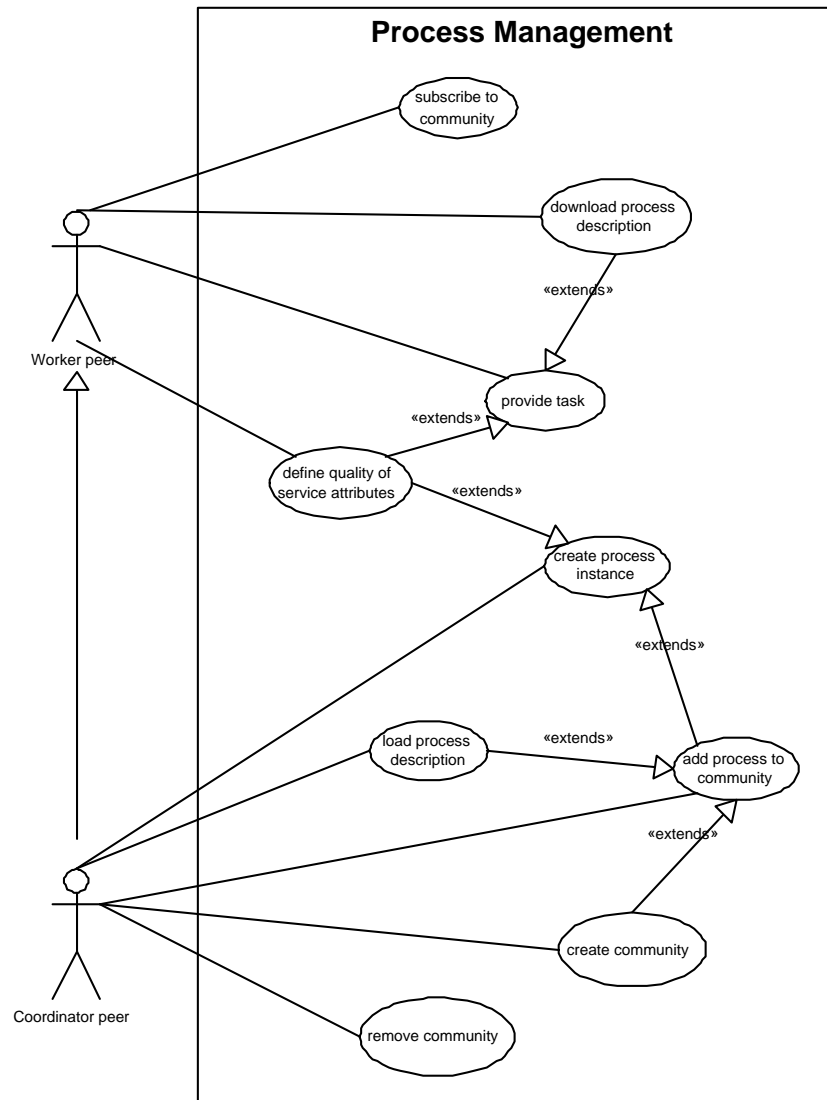


Figure 4.2: Task status

**Create community** A coordinator can establish a community which will be the communication platform for a process or a group of related processes he intends to manage. The name of the new community can be specified by the user. All members of the workflow system are informed about the new community and can subscribe it to receive the messages.

**Remove community** A community which is not required anymore can be removed from the system. The peers are informed about this and remove the references from their local databases. All task announcements of this community are also removed.

**Load process description** A new process description which is still unknown to the system can be imported from a file. It has to contain the description in BPEL format and must be available on the local file storage. The new process is added to the local repository. The Workflow Coordinator also creates an artifact with this file, which can be downloaded by other peers.

**Add process to a community** To make a process visible to other peers, it has to be added to a process community. Members of this community are informed about the new process and can retrieve the artifact containing the description file.

**Create process instance** To execute a process a coordinator can create a process instance. This will create an instance community which will serve as an environment for communication and document sharing. An instance can only be created for a process which has been added to a process community.

**Download process description** A peer who wants to participate in a workflow instance and provide a task, needs to have the process description available in the local database. If the description is not known to the peer, he can request an artifact containing the required BPEL document from the coordinator. After the successful download it is imported into the local repository.

**Provide task** A task can be provided to a process community if it is part of a process known to the peer. It is necessary, that the peer has subscribed to that community prior to announcing the task.

**Define Quality of Service attributes** On providing a task, the peer has to give quality of service attributes which define the terms on which it will be carried out. The attributes help the coordinator to compare announcements of different peers.

**Subscribe to community** A peer which wants to become member of a process community can subscribe to it. This is necessary to receive any messages concerning the community, like task announcements.

## 4.2 Task Negotiation

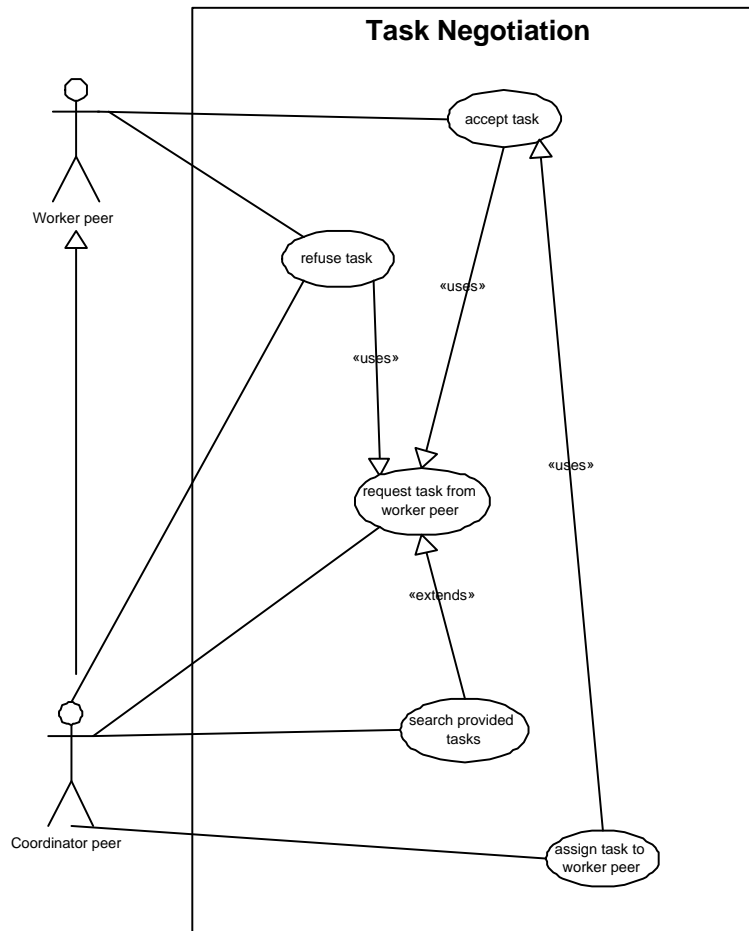


Figure 4.3: Task Negotiation

**Search provided tasks** The coordinator can actively query all peers of a task he needs. Peers which provide that task re-send an announcement to the sender. This can be necessary for peers which have been disconnected for some time and want to update their local repositories.

**Request task from peer** If the coordinator wants to give a task of an instance to a user, he looks up the database for peers which provide this task. To elect one of them, he can compare the quality of service attributes given on announcing the task. The coordinator then sends a task request to the specific peer.

**Accept task** A worker peer which has received a task request has to decide whether he wants to accept the task or not. An acceptance sends a message to the coordinator, indicating that the peer wants to execute the task.

**Refuse task** Both worker and coordinator peer have the possibility to refuse a task. The former if he doesn't want to serve a task and the latter if he admittedly had requested the task, nevertheless doesn't want to assign that peer.

**Assign task** If a worker peer has accepted a task request, the coordinator can assign him to definitely perform the work. The peer receives the information concerning the instance community the task is performed in. He has to subscribe to the instance community in order to have access to any relevant documents.

### 4.3 Instance Execution

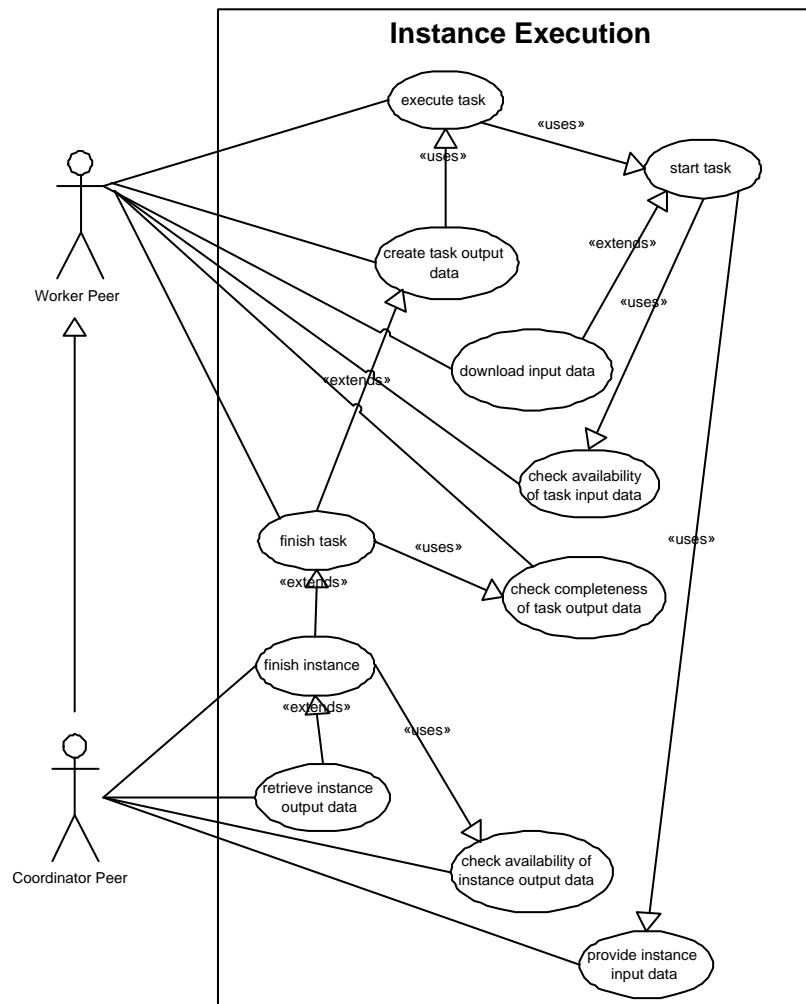


Figure 4.4: Instance Execution

**Provide instance input data** The coordinator has to ensure, that the worker peers have access to the input documents, which are required according to the process description. Before the peers can start the work, the coordinator has set a document for each input container. The workflow system wraps the documents into artifacts and publishes them to the instance community where they can be downloaded from.

**Check availability of task input data** A peer which wants to start to execute a task has to check, whether it requires any input data. If applicable,

the peer has to query the community whether these documents have already been submitted.

**Download input data** After having successfully checked the availability of input data, the artifacts representing the input documents can be downloaded to the local repository.

**Start task** As soon as all requirements are met, the worker peer can start the task. This primarily includes that the task has been assigned to the peer and that all required documents have been downloaded. The coordinator will be informed about the new status.

**Execute task** The worker peer has to perform the work, which is associated with the task. Usually he has to process the input documents and create output documents as required. The task must have been started prior to the execution.

**Create task output data** After having created the necessary output documents in the file storage, they have to be made available in the instance community. Similar to the provision of input data, the documents were wrapped into artifacts and published to the instance community.

**Check completeness of task output data** Before a task can be finished, it must be ensured that all required output documents have been successfully delivered. The peer queries the instance community whether all output containers are available.

**Finish task** Having delivered the output documents and successfully checked for completeness, the status of the task is set to finished. The peer informs the coordinator about the successful execution.

**Check availability of instance output data** The coordinator can check whether the output documents have been submitted by the worker peers. He looks up the instance community whether participating peers have published the corresponding artifacts.

**Retrieve instance output data** If the coordinator has successfully checked the availability of the documents, he downloads the artifacts representing the output containers.

**Finish instance** When all peers have fulfilled their tasks and the output data is collected, the instance will be set to the final state by the coordinator. The instance community will be deleted as the last step in the execution of the process.

## Chapter 5

# Implementation

In this chapter we focus on implementation details of the workflow system. Based on the design issues discussed in chapter 3, the development was performed using the Java2 platform. The core classes are available in the package `eu.motion.tuv.workflow`, integrated in the MOTION middleware.

First we give an overview of the classes and outline the functionality and usage of the most important methods. Then we describe the structure of the process description files and will continue with the internal representation of the database files. The next section defines the various message types the peer communication relies on. After that we focus on the different possibilities to exchange documents between peers.

### 5.1 Class diagram

This section gives an overview of the classes of the workflow system. We will outline the usage of the most important ones and give a short description of attributes and methods. A detailed relationship diagram is shown in figure 5.1.

#### 5.1.1 WFCoordinator

It is the main class in the system and provides an interface for the application layer. It is responsible for sending messages using the MOTION middleware and gives access to the local database. The methods of this class are non-blocking, as the `WFCoordinator` doesn't wait for replies on the messages sent. Incoming messages are trapped and handled by the class `WFDatabase`.

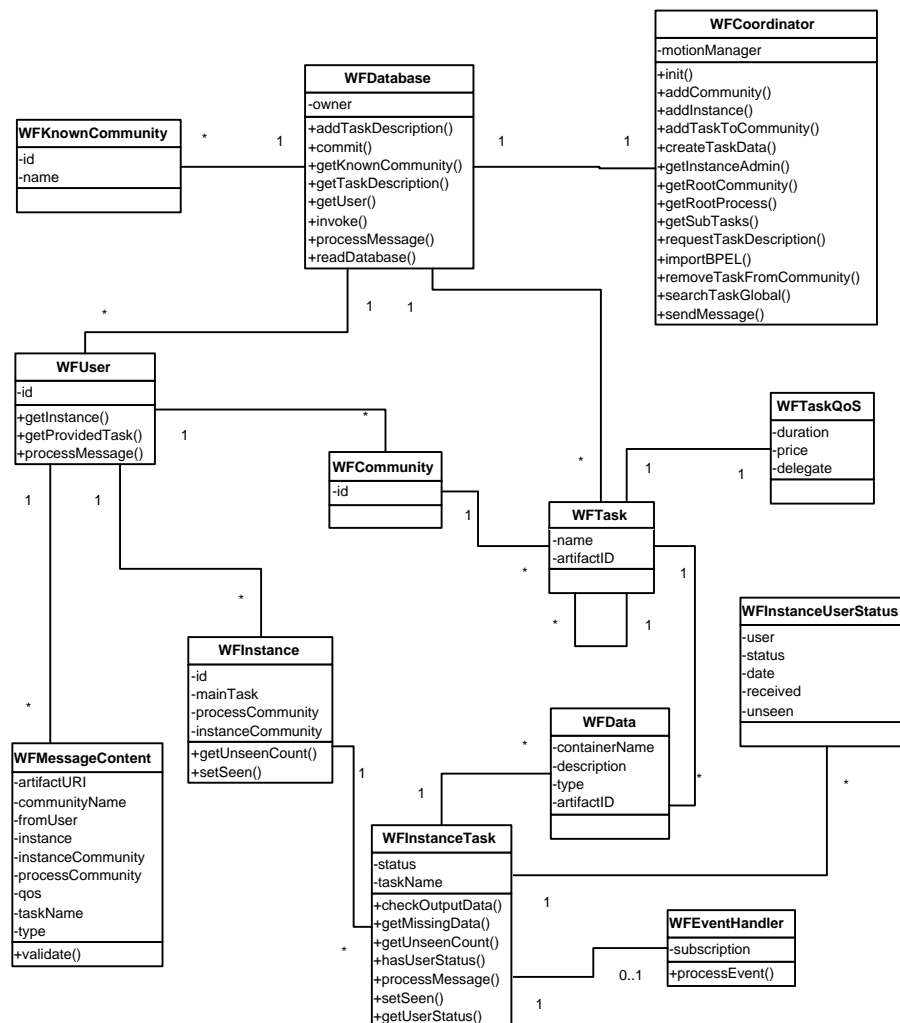


Figure 5.1: Class diagram

- The `addCommunity()` method is used to add a process community to the system. It creates a community in MOTION with the given name and sends a message to all peers about the details.
- `addTaskToCommunity()` publishes a given task or process to a community. The members of the community are notified of the new task announcement.
- On execution of a task, a peer may have to submit output documents according to the process description. The `createTaskData()` method provides the functionality to assign a document to a specific data container. The given file is wrapped into a MOTION artifact and submitted to the instance community.

- `getInstanceAdmin()` returns the user which is the coordinator of a given instance.
- The method `getRootCommunity()` returns the top-level community *WFCommunity*, which is mainly used as a target of global messages.
- Given the name of a task, `getRootProcess()` looks up the process description table and returns the process the task is part of.
- If a peer receives a message concerning an unknown task, he can request the description from the sender using `requestTaskDescription()`. A message is sent to the corresponding user, asking him to send the corresponding artifact.
- To add a new process description to the system, the `importBPEL()` method parses the given BPEL-file and adds it to the local database.
- Invoking `searchTaskGlobal()`, a peer can actively search for all providers of a specific task.
- The method `sendMessage()` provides the functionality of sending messages using the MOTION middleware. Message type and content are specified by a `WFMessageContent` object, which is passed as an argument.

### 5.1.2 WFDatabase

This class is responsible for the management of users, processes, and instances and is accessed by the `WFCoordinator` class. It provides methods to access the objects which are available in the local database files. On startup of the system, it reads the XML database file of the current user and holds the information in memory. To maintain a persistent state of the data, all methods of this class commit the changes to the local storage immediately.

The `WFDatabase` also listens to MOTION messages and processes them upon reception.

- The method `commit()` writes the database of the current user to a persistent file using the PDOM component.
- Given the name of a task, `getTaskDescription()` returns the description of the task.
- As this class is registered to receive MOTION messages, the `invoke()` method is called whenever a message from another peer arrives. The content is parsed and passed to `processMessage()`.

- Arriving messages are handled in the method `processMessage()`. If the content is specific to a certain user, the appropriate `WFUser` object will proceed to process the information.
- To initially load the local database file, `readDatabase()` is invoked on startup. It parses the XML file using the XQL component and creates the necessary objects in memory.

### 5.1.3 WFTask, WFData, WFTaskQoS

As the internal data representation makes no difference between processes and tasks, they are represented by the same class, `WFTask`. The semantic to distinguish between them is, that a `WFTask` object is considered to be a process if it is no subtask of any other task.

The attribute `artifactID` is only relevant for the process representation. It contains the ID of the artifact containing the process description file.

The `WFData` class implements the data container concept. It gives access to the documents which have been associated with the container. It is identified by the `containerName` attribute. If an artifact has already been created, its ID is stored in the attribute `artifactID`.

To store quality of service attributes of provided tasks, the class `WFTaskQoS` contains the relevant data structure. We have three predefined attributes here: The `duration` specifies the expected time to execute the task, the `price` shows how much will be charged for consuming the service, and `delegate` is a flag, which indicates whether the peer himself executes the task or it will be delegates to a third party.

### 5.1.4 WFUser

For every user known to the workflow system, a `WFUser` object is available in the database. It contains a list of instances the user coordinates and has a list of all tasks the user provides. The tasks are grouped by the community they have been announced in.

- Calling the method `getProvidedTask()` with a process community and a task name as arguments we receive a `WFTask` object which contains the details of the task as it was announced.
- The method `processMessage()` is called by the `WFDatabase` if a received message is specific to the user which is represented by this `WFUser` object. It analyzes the content and updates the relevant objects of the database.

### 5.1.5 WFInstance

This class contains the relevant information about instances the user is involved in. It saves a reference to the instantiated process in the attribute `mainTask`. Other important attributes are information about the community the process has been published in (`processCommunity`) and the private instance community (`instanceCommunity`).

### 5.1.6 WFInstanceTask

For each task of the instance the user is involved in, a `WFInstanceTask` object is stored in the database. The input and output containers which have to be considered when executing the task are stored in a list of `WFData` objects.

- The method `checkOutputData()` iterates all output containers and returns, whether all artifacts have been specified and the task is allowed to set its state to finished.
- All input containers are checked with the method `getMissingData()`. If there are input documents missing, it searches the community for corresponding artifacts and downloads them, if available. Otherwise it creates a MOTION subscription with criteria matching the desired artifact.

### 5.1.7 WFEventHadler

The class `WFEventHandler` is an event handler, which listens to subscriptions created in the method `WFInstanceTask.getMissingData()`. As soon as an artifact has been created which matches the subscription, the `processEvent()` method is invoked. It then automatically downloads the desired artifact and updates the `WFData` object which it refers to.

## 5.2 Process description and BPEL

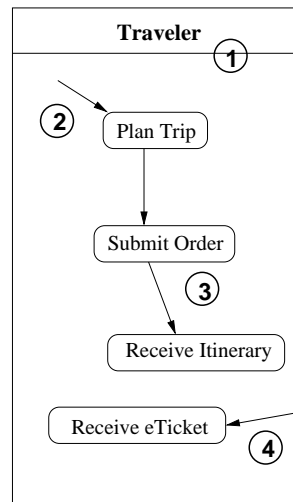


Figure 5.2: BPEL example graph

The detailed description of processes have to be provided in an XML file which complies with the BPEL specification in [31]. Nevertheless, the requirements for our workflow system are not as sophisticated as the complete language definition. For that reason we don't use all BPEL features but work with a limited subset of it.

In this section we will show how the process description is built according to a graphical representation of the workflow. For simplicity reasons we have taken some tasks of the total business process as illustrated in figure 5.2. As the the original description doesn't need any input documents, we have added one to also show this functionality: We assume, that the coordinator has to provide a process input document, called *catalogue* which the traveler requires to start planning the trip. This is indicated by an arc denoted as (2) in the illustration.

The process description in BPEL notation which we refer to in the next paragraphs is shown in the listing in figure 5.3. As this is not the complete file, we use dots (...) as a placeholder for missing parts.

### 5.2.1 Structure

A BPEL definition consists of three major sections, `<partners>`, `<containers>`, and `<flow>`. The first two are declarative and describe roles and objects, and the third one gives the tasks and relations between them.

#### *Partners, line 2-6.*

This section defines the different parties which interact with the business process when it is executed. The three partners shown here correspond to the person who wants to have his trip planned (*traveler*), as well as the *agent* which processes his request, and an *airline* which does the actual flight and seat reservation. The workflow graph shows the partners in the top section, denoted with (1). In practice, a partner role defines a set of tasks which reasonably will be performed by one person or peer. For example, in an instance of our airline reservation process, there will practically only be one peer which performs the tasks *Plan Trip*, *Submit Order*, *Receive Itinerary*, and *Receive Ticket*. Anyway, as the semantical context is considered out of scope in our workflow system, we ignore any influence of the partners definition on the execution course of a business process.

#### *Containers, line 8-13.*

This declaration defines the containers which serve for data exchange between the tasks and give access to process input and output documents. The different containers, which are necessary throughout a process execution, have to be identified during the requirement analysis of the business process. Containers have also the function of determining dependencies between the tasks. We define, that a task B is dependent on a task A, if task A stores documents into a container which are required for task B's execution.

#### *Flows, line 16-45.*

In the third part of the process definition we have the actual task definitions. This includes information about how they are linked together and which containers are passed between them. To comply with all kinds of document flows we have to deal with, we have to distinguish between three flow types which are identified as follows:

- **Process input data.** A process input document which has to be provided by the coordinator is denoted with (2) in figure 5.2. The implementation in BPEL is done with a `<receive>` node (line 17-20). Here, the documents have to be provided in the container *catalogue*.

- **Task.** The tasks of the process are represented by `<invoke>` nodes, with the task name given in the `operation` attribute. The input and output documents are described in the attributes `inputContainer` and `outputContainer` respectively. Arcs, which point from one operation to another and thus show a connection between them, are mapped to links: For example, the tail of the arc number (3) is represented by a link source in line number 25, and the the head of that arc is connected to the task *Submit Order* by a link target in line number 32, both identified by the arc's name *planTrip-ready*.
- **Process output data.** A process can be considered as finished, when all output containers are provided. In our example this is done, when the traveler has received his tickets, denoted as (4) in the illustration. The coordinator can then retrieve the documents. The lines 39-41 in the BPEL file show the `<reply>` node which delivers the container *issuedTickets* to the coordinator.

```

1 <process name="airline">
2   <partners>
3     <partner name="Coordinator" />
4     <partner name="Traveler" />
5     <partner name="Agent" />
6     <partner name="Airline" />
7   </partners>
8
9   <containers>
10    <container name="locationsAndDates" />
11    <container name="submittedOrder" />
12    ...
13    <container name="issuedTicktes" />
14  </containers>
15
16  <flow>
17    <receive partner="Coordinator"
18      container="catalogue">
19    </receive>
20
21    <invoke partner="Traveler"
22      operation="planTrip"
23      inputContainer="catalogue"
24      outputContainer="locationsAndDates">
25      <source linkName="planTrip-ready" />
26    </invoke>
27
28    <invoke partner="Traveler"
29      operation="submitOrder"
30      inputContainer="locationsAndDates"
31      outputContainer="submittedOrder">
32      <target linkName="planTrip-ready" />
33      <source linkName="submitOrder-ready" />
34    </invoke>
35
36    .....
37

```

```
38  
39     <reply partner="Traveler"  
40           container="issuedTicktes">  
41     </reply>  
42  
43 </flow>  
44 </process>
```

Figure 5.3: BPEL definition of the airline process

### 5.3 Internal database format

The local data storage of the peers is implemented using PDOM and XQL [38]. The database contains any information which is relevant for process management, as well as messages received from and sent to other peers. Every peer has its own database which gives the peer's local view of the workflow engine. This section will give an overview of the data structure which is listed in figure 5.4.

The content of the database has three main data sections (line 3). The `<Description>` node contains process definitions which have either been imported or downloaded, and the `<User>` nodes contain any information received from and sent to a specific peer. The `<KnownCommunity>` nodes list all communities known to the peer.

**Description node.** All tasks known to a peer are stored as `<Task>` nodes here (line 8-9). This includes default values for quality of service attributes and the definition of flow data, which is referred to as *container* in BPEL notation.

**User nodes.** The user nodes (line 27) represent the knowledge base of the communication with other peers. It contains task announcements and details about participations in process instances. Every peer which had a message delivered to the owner of the database is represented with a `<User>` node. Furthermore the owner of the data storage himself has node, containing its own announcements and participations.

- The `<Provide>` node (line 30) contains all tasks which are provided by the specific peer, grouped by the community they have been announced in.
- The most important nodes for process executions are the `<WorkUnits>` (line 35): All instances a peer is involved in are listed here. In more detail, the `<InstanceTask>` subnodes (line 43) describe the messaging which has taken place in this context. The exchange of documents which has taken place between the participants of the instance is also tracked in this subtree. Containers which have already been provided to the specific instance are represented by the artifact, stored in the `<WFData>` child nodes.

**KnownCommunity nodes.** All communities, which have been added to the workflow system are listed here. This does not include instance communities, which are only visible to participants of the specific instances.

```

1  <!DOCTYPE WFDatabase [
2
3  <!ELEMENT WFDatabase (Description,User*,KnownCommunity*)>
4  <!ATTLIST WFDatabase owner CDATA #REQUIRED>
5
6  <!ELEMENT Description (Task*)>
7
8  <!ELEMENT Task (Subtask*, QoS, FlowData*)>
9  <!ATTLIST Task name CDATA #REQUIRED>
10
11 <!ELEMENT Subtask>
12 <!ATTLIST Subtask CDATA name #REQUIRED>
13
14 <!ELEMENT FlowData (Comment)>
15 <!ATTLIST FlowData name CDATA #REQUIRED>
16 <!ATTLIST FlowData type CDATA #REQUIRED>
17 <!ATTLIST FlowData artifact CDATA #IMPLIED>
18
19 <!ELEMENT Comment (#PCDATA)>
20
21 <!ELEMENT QoS (duration,price,delegate)>
22
23 <!ELEMENT duration (#PCDATA)>
24 <!ELEMENT price (#PCDATA)>
25 <!ELEMENT delegate (#PCDATA)>
26
27 <!ELEMENT User (Provide, WorkUnits)>
28 <!ATTLIST User id CDATA #REQUIRED>
29
30 <!ELEMENT Provide (Community*)>
31
32 <!ELEMENT Community (Task*)>
33 <!ATTLIST Community id CDATA #REQUIRED>
34
35 <!ELEMENT WorkUnits (Instance*)>
36
37 <!ELEMENT Instance (InstanceTask+)>
38 <!ATTLIST Instance id CDATA #REQUIRED>
39 <!ATTLIST Instance mainTask CDATA #REQUIRED>
40 <!ATTLIST Instance processCommunity CDATA #REQUIRED>
41 <!ATTLIST Instance instanceCommunity CDATA #REQUIRED>
42
43 <!ELEMENT InstanceTask (Message*, FlowData*)>
44 <!ATTLIST InstanceTask taskName CDATA #REQUIRED>
45 <!ATTLIST InstanceTask processCommunity CDATA #REQUIRED>
46 <!ATTLIST InstanceTask instanceCommunity CDATA #REQUIRED>
47
48 <!ELEMENT Message>
49 <!ATTLIST Message date CDATA #REQUIRED>
50 <!ATTLIST Message received CDATA #REQUIRED>
51 <!ATTLIST Message user CDATA #REQUIRED>
52 <!ATTLIST Message type CDATA #REQUIRED>
53
54 <!ELEMENT KnownCommunity>
55 <!ATTLIST KnownCommunity id CDATA #REQUIRED>
56 <!ATTLIST KnownCommunity name CDATA #REQUIRED>
57
58 ]>

```

Figure 5.4: DTD of local database

## 5.4 Peer messaging

The exchange of information between the participants of the workflow system is mostly performed using messages. This communication is implemented using MOTION messages. They can be targeted to a specific peer or a community, respectively. In our implementation both mechanisms are used, depending on the requirements of the information being transmitted.

### 5.4.1 Announce a new community

<b>Message</b>	WF_NEW_COMMUNITY
<b>Target</b>	WFCommunity
<b>Parameter</b>	<i>id</i> : MOTION id of the new community <i>name</i> : Name of the new community
<b>Description</b>	When a user creates a new Process Community, he has to input a name for it. The Workflow Coordinator then creates a MOTION Community and receives a unique id. The message containing both the <i>name</i> and the <i>id</i> is sent to the WFCommunity and delivered to all peers. The recipients automatically add the information to their local database.

### 5.4.2 Remove a community

<b>Message</b>	WF_REMOVE_COMMUNITY
<b>Target</b>	WFCommunity
<b>Parameter</b>	<i>id</i> : MOTION id of the community
<b>Description</b>	If a user decides to remove a Process Community, all peers have to be informed immediately. The chosen community is removed from the middleware and a message containing the <i>id</i> is sent to the WFCommunity. The peers automatically remove the community from their databases. This of course implies, that any task announcements of this community becomes invalid and is removed as well.

### 5.4.3 Global search for a task

<b>Message</b>	WF_SEARCH_TASK
<b>Target</b>	WFCommunity
<b>Parameter</b>	<i>taskName</i> : Name of the task to search
<b>Description</b>	The WF_SEARCH_TASK message is the only message which actively queries other peers whether they provide a task. The content includes the name of the desired task. Peers which receive this message look up their own databases if they have announced the given task in any community. If one currently provides the task, it replies to the sender with a WF_PROVIDE_TASK message, containing the relevant details. As this method of querying for announcements is rather expensive in terms of bandwidth use, it should not be used frequently. It is meant to update a peers database when reconnecting to the workflow system after a long time.

### 5.4.4 Announce a task

<b>Message</b>	WF_PROVIDE_TASK
<b>Target</b>	Process Community
<b>Parameter</b>	<i>taskName</i> : Name of the task <i>processCommunityId</i> : Community the task is published in <i>qos</i> : Quality of Service attributes
<b>Description</b>	If a peer wants to provide a task, it has to choose a task, select the Process Community, and specify the Quality of Service attributes. This information is included in the message and transmitted to the subscribers of the chosen community. The recipients have to update their local databases. If a peer finds that the sender has published this task before, it removes the old announcement and replaces it with the new one. This results in an update of the Quality of Service attributes.

### 5.4.5 Revoke a task announcement

<b>Message</b>	WF_UNPROVIDE_TASK
<b>Target</b>	Process Community
<b>Parameter</b>	<i>taskName</i> : Name of the task <i>processCommunityId</i> : Community of the task
<b>Description</b>	To revoke a task announcement a peer uses this message. The former announcement is identified by the name of the task ( <i>taskName</i> ) and the Process Community the task was published in ( <i>processCommunityId</i> ). The recipients of this message query their databases and remove the relevant entries if existent.

### 5.4.6 Request the process description for a task

<b>Message</b>	WF_GET_TASK_DESC
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName</i> : Name of the task
<b>Description</b>	A peer which receives an announcement of a task it has no information of in its database, it can request the process description file for this task. The message is transmitted to the sender of the task announcement with the <i>taskName</i> as argument. When a peer receives this request it looks up its database which process it concerns and replies the relevant information using a WF_SEND_TASK_DESC message.

### 5.4.7 Send process description

<b>Message</b>	WF_SEND_TASK_DESC
<b>Target</b>	Peer
<b>Parameter</b>	<i>artifactUri</i> : Artifact URI of the process description file
<b>Description</b>	This message is a reply on a WF_GET_TASK_DESC request and includes the resource identifier ( <i>artifactUri</i> ) for a MOTION artifact which contains the process description file. The receiver of such a message automatically downloads the artifact from the given URI and imports the process description file into its database. The URI itself will also be stored in order to being able to serve requests from other peers. The download is illustrated in figure 3.5.

### 5.4.8 Request a task

<b>Message</b>	WF_REQUEST
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName</i> : Name of the task <i>instanceId</i> : id of the process instance <i>processCommunityId</i> : id of the Process Community <i>instanceCommunityId</i> : id of the Instance Community
<b>Description</b>	<p>This message is sent to ask a peer whether it wants to execute the given task. The parameters <i>taskName</i> and <i>processCommunityId</i> are used by the recipient to identify the specific task the sender refers to. This is important, because the peer can have announced a task in several Process Communities with different Quality of Service attributes. The parameters <i>instanceId</i> and <i>instanceCommunityId</i> refer to the process instance the task should be performed in. The request is added to the local database and marked as a new message, but no further action is taken.</p>

### 5.4.9 Accept a task request

<b>Message</b>	WF_REPLY_ACCEPT
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName</i> : Name of the task <i>instanceId</i> : id of the process instance <i>processCommunityId</i> : id of the Process Community
<b>Description</b>	<p>To reply to a WF_REQUEST message and inform the sender that the task is ready to be executed, a peer has to send this message. The parameters included help the requesting peer to match the reply to his prior message.</p>

### 5.4.10 Refuse a task request

<b>Message</b>	WF_REPLY_REFUSE
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName</i> : Name of the task <i>instanceId</i> : id of the process instance <i>processCommunityId</i> : id of the Process Community
<b>Description</b>	<p>This message can be sent in two different cases: (a) a peer which receives a WF_REQUEST message doesn't want to perform the task, or (b) a requesting peer gets a positive answer but decides at this point of the negotiation that it will not entrust it with this work. The parameters included help the requesting peer to match the reply to its prior message.</p>

### 5.4.11 Assign a task to a peer

<b>Message</b>	WF_ASSIGN
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName:</i> Name of the task <i>instanceId:</i> id of the process instance <i>processCommunityId:</i> id of the Process Community <i>instanceCommunityId:</i> id of the Instance Community
<b>Description</b>	This message informs a peer to actually perform the task. Upon reception of this notification the peer has to subscribe to the instance community identified by the parameter <i>instanceCommunityId</i> in order to receive all messages concerning the instance.

### 5.4.12 Start execution of a task

<b>Message</b>	WF_REPLY_START
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName:</i> Name of the task <i>instanceId:</i> id of the process instance <i>processCommunityId:</i> id of the Process Community <i>instanceCommunityId:</i> id of the Instance Community
<b>Description</b>	This message informs the coordinator that all requirements (e.g. availability of input data) have been met to start the task. The coordinator stores this information into its database but doesn't take any further action. The purpose of this notification is to keep track of the current instance status.

### 5.4.13 Finish execution of a task

<b>Message</b>	WF_REPLY_FINISH
<b>Target</b>	Peer
<b>Parameter</b>	<i>taskName:</i> Name of the task <i>instanceId:</i> id of the process instance <i>processCommunityId:</i> id of the Process Community
<b>Description</b>	This message informs the coordinator that the task identified by the parameter has been successfully finished. The coordinator updates its database and checks the overall instance status. If every single task is finished, the output data can be retrieved by the coordinator.

## 5.5 Exchange of documents

Besides communicating using messages, the workflow system requires a method to exchange data files. The MOTION middleware supports us with the artifact concept which provides a mechanism to share files among different peers. An artifact represents an object which can be published to a community. It is then visible to all members of that community and can be queried by other peers using meta information which is attached to the artifact. We have to consider two document types which have to be retrieved differently: Process description files and instance documents.

### 5.5.1 Process description files

A peer which wants to download a process description file has to request it from the coordinator using messages (see section 5.4). A successful reply contains the URI of the desired file. Using the MOTION middleware the document identified by the URI can be downloaded. These steps are illustrated in figure 5.5.

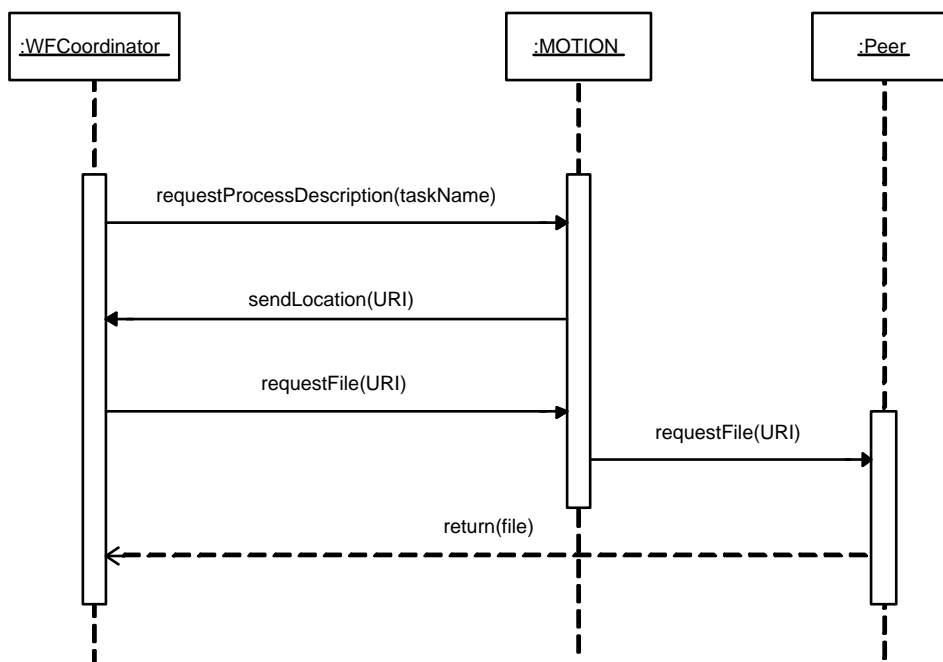


Figure 5.5: Download a process description from another peer

### 5.5.2 Instance documents

During instance execution peers may have to produce output data to fulfill a task they are assigned to perform. These files have to be added to the workflow engine that other peers, which require this information, can retrieve it. Which documents are involved in a task is specified in the data container definition of the process description. The document of a data container will be stored as an artifact as soon as it becomes available. It will be added to the corresponding instance community for other peers to find it.

In order to being able to identify an artifact, we need a matching between artifacts and data container. This is established using the `NAME` attribute of the artifact, which contains the name of the corresponding data container. When a peer needs a specific artifact, it can look up the list of artifacts of the instance community and match the `NAME` attribute with the name of the data container which it desires.

### Deliver a document for an instance task

To deliver a document, a user has to choose an instance, and a task which he is assigned to perform, and the data container he wants to submit. Furthermore a file has to be chosen which contains the data of interest. The WFCoordinator then will create an artifact for the given file and set the name according to the data container. The artifact is added to the instance community and the data container in the local database is updated with the id of the artifact (see figure 5.6).

The same steps apply when a coordinator wants to execute an instance which, according to the process description, requires some input data.

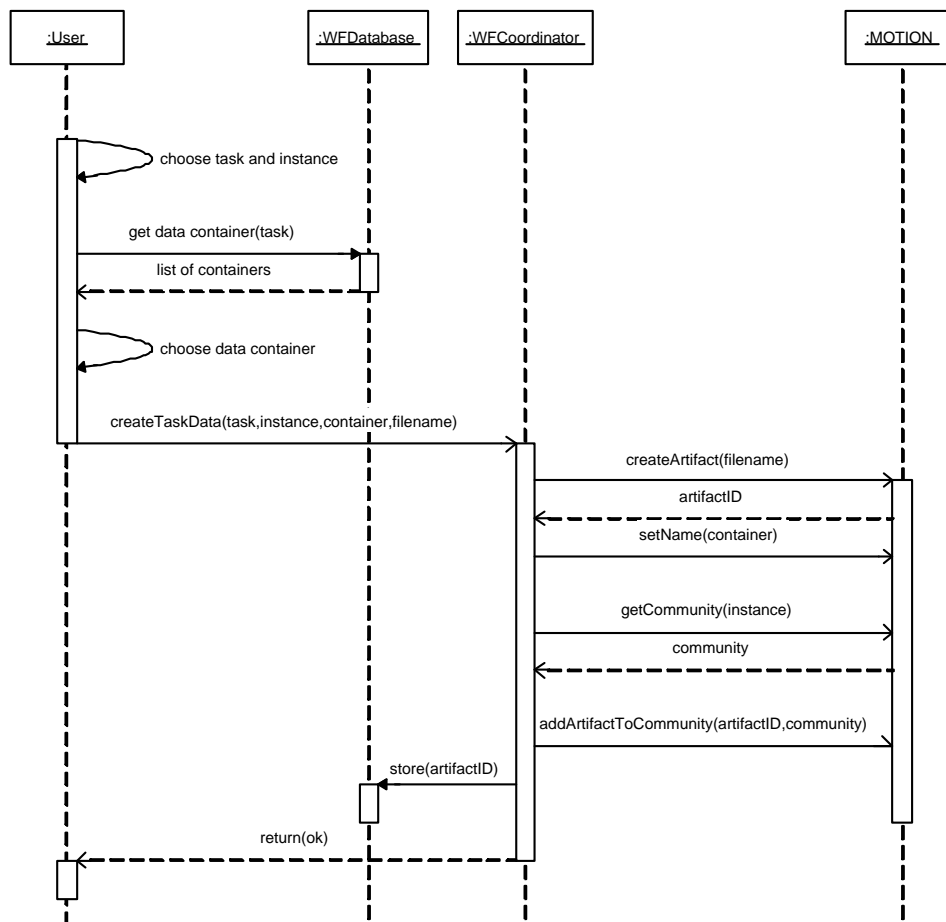


Figure 5.6: Deliver a document for an instance task

**Retrieve documents of an instance task**

When a peer wants to start the execution of a task, it has to ensure that all input data the task depends on is available in the instance community. To do so, the peer looks up the process description for all data containers of the corresponding task. For each the Workflow Coordinator searches for artifacts in the community, matching the `NAME` attribute with the specific container name. If it is not available, which means that the desired document has not been submitted to the community, the workflow coordinator automatically subscribes on this search criteria. As soon as any peer creates the desired artifact, a notification is fired which causes the workflow coordinator to retrieve the artifact. Using the included URI the document can be downloaded from the corresponding peer. A detailed sequence of these steps are outlined in figure 5.7.

When a coordinator executes an instance which as a result has output data, he can retrieve the documents the same way as described here.

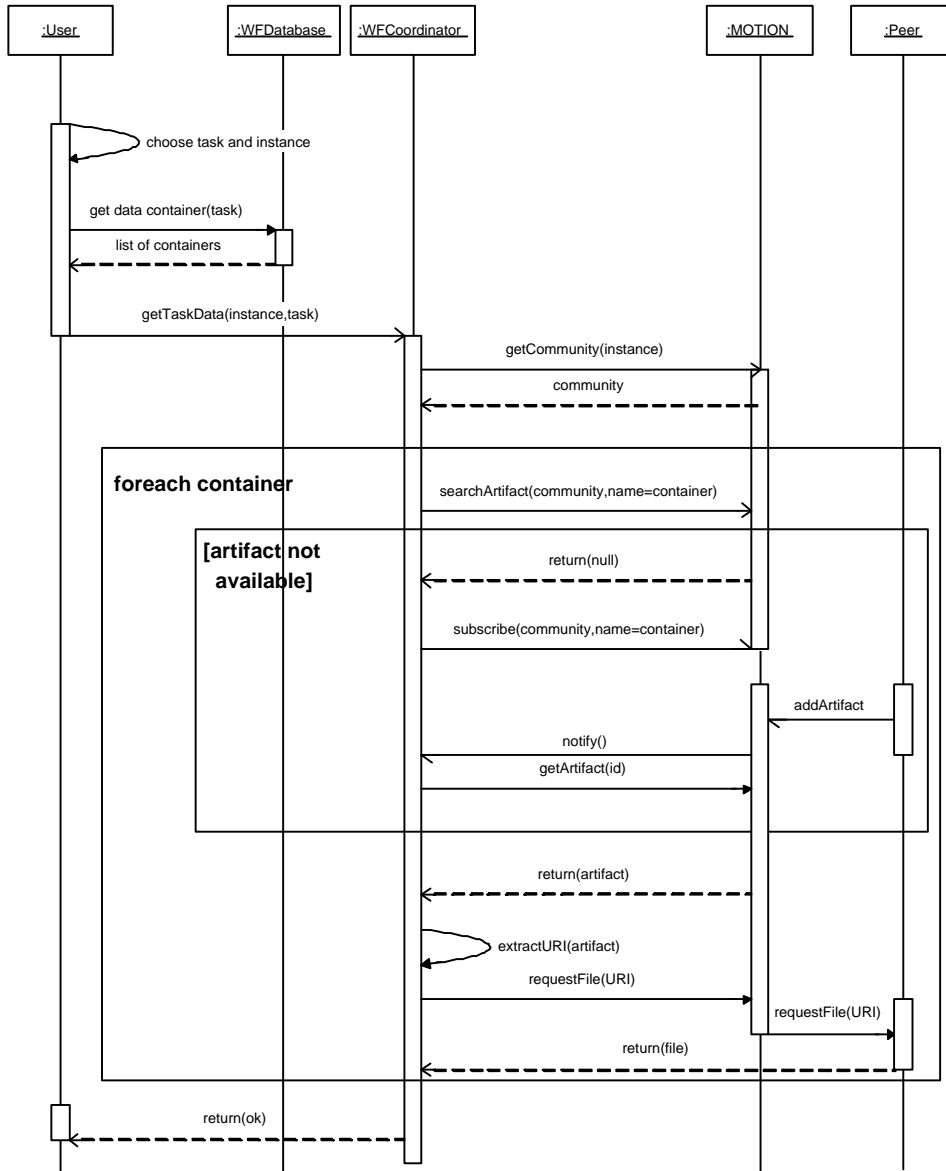


Figure 5.7: Retrieve documents of an instance task

## Chapter 6

# Validation

In this chapter we want to show, how the functionality of the workflow system has been integrated into a graphical user interface. The GUI is an extension of the the Java MOTION client and introduces three modules:

- The *Process Manager* basically supplies us with the functionality of community management, handling of process descriptions, and providing tasks.
- The *User Manager* shows the current view of all known users and provided tasks and contains the process description download.
- In the *Instance Overview* task negotiations and instance executions take place. Furthermore, it has a sub-module *Instance Manager* which the coordinator can use to supervise and coordinate instances.

Taking into account our airline reservation case study, the next sections show various steps of process management issues and the execution of an instance. We have two peers involved, one acting as the coordinator (user name admin) and the other acting as a worker (user name data).

## 6.1 Process Management

This section describes the steps which are necessary to set up an environment for the execution of a process. The coordinator peer is responsible for creating a community, adding a process to the workflow system, adding the process to the newly created community and creating an instance for the process. The worker peer which participate in this workflow has to join the specific process community. Then he can download the process description and choose tasks which he provides for execution. Figure 6.1 gives an overview of these steps.

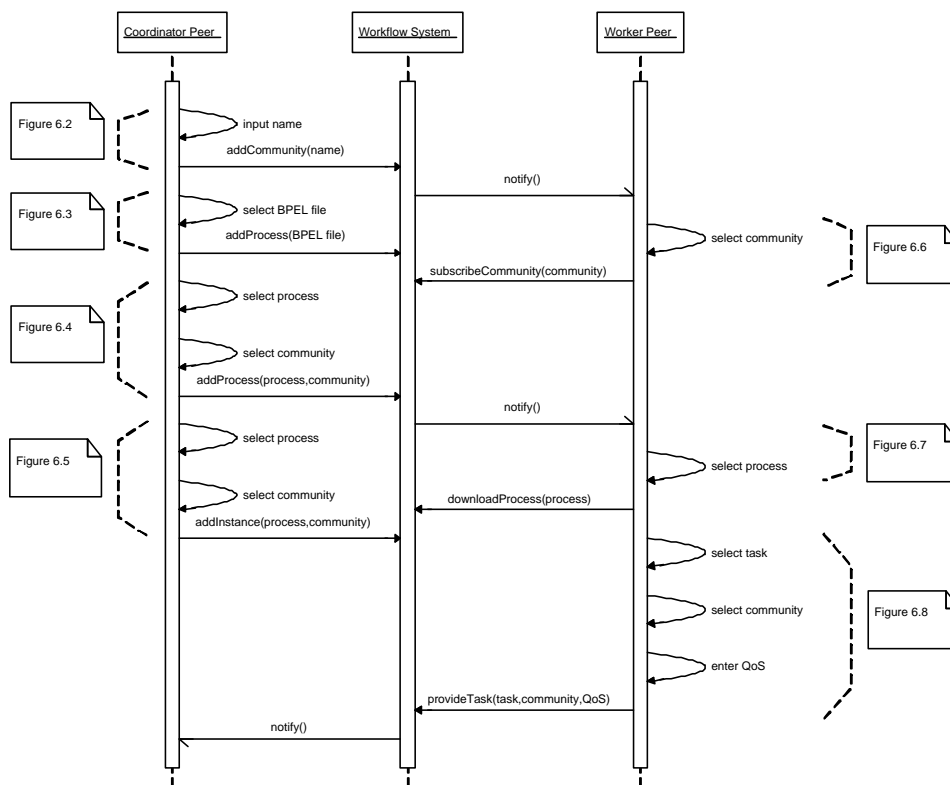


Figure 6.1: Process Management overview

### 6.1.1 Coordinator: Add a community

The creation of a community takes place in the *Workflow Process Manager*. On choosing the appropriate menu item, the user is asked for the new community's name. The new community is then visible to all users of the network.

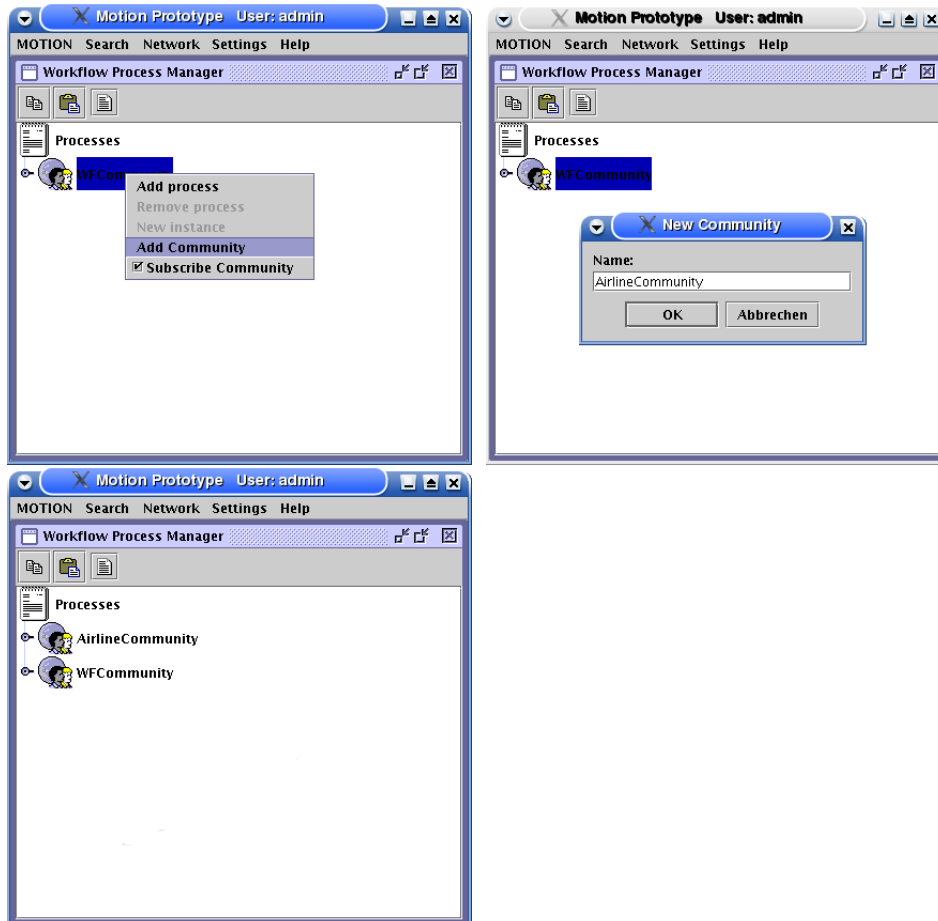


Figure 6.2: Add a community

### 6.1.2 Coordinator: Add process description

To import a process description to the workflow system we need to load it from a local file in BPEL-format. The process with all tasks will be shown in the *Workflow Process Manager* next to the available communities.

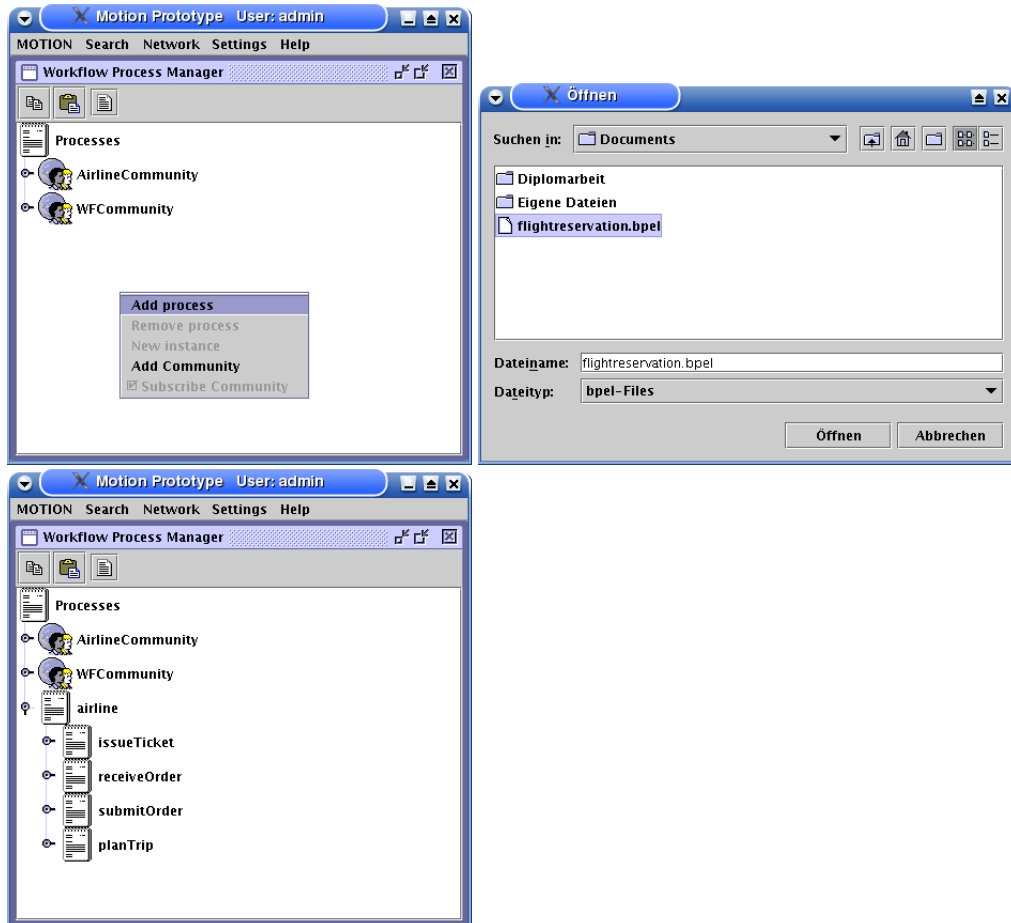


Figure 6.3: Add process description

### 6.1.3 Coordinator: Add process to a community

So far, the description is only available locally. To make it accessible to other peers, we have to publish it to a process community. This is performed in the *Workflow Process Manager* using a copy and paste concept. We highlight the airline process and click the *Copy* button in the toolbar. Then we mark the *AirlineCommunity* and use the *Paste* button to publish it to the selected community.

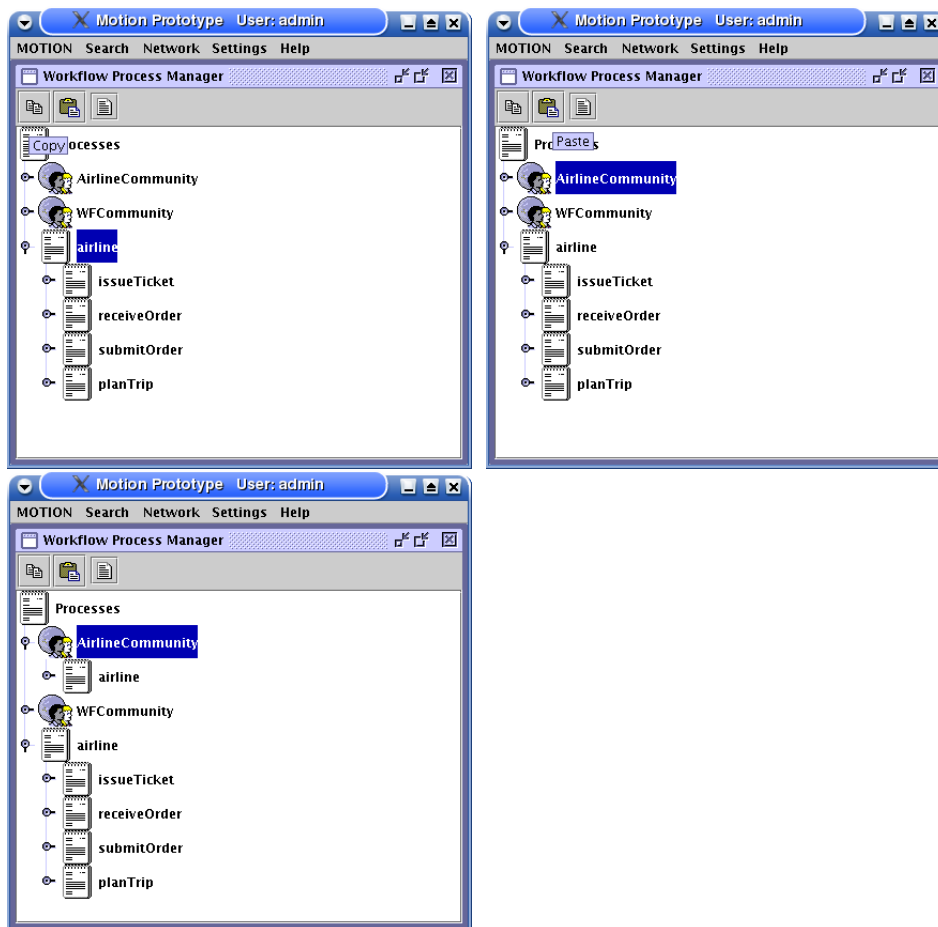


Figure 6.4: Add a process to a community

### 6.1.4 Coordinator: Create a process instance

To create an instance and set up an environment for the execution, we have to highlight the `airline` process and choose the appropriate item from the context menu. The details of the instance can be seen in the *Instance Manager* which will be shown in the next section.

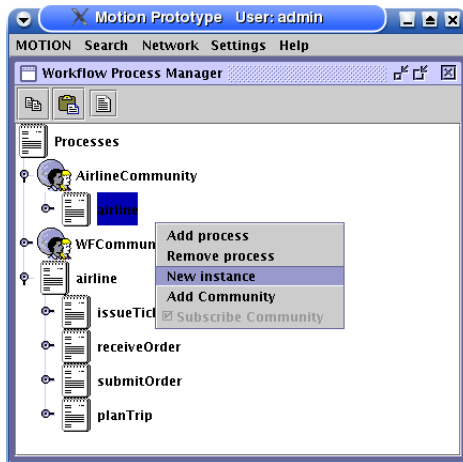


Figure 6.5: Create an instance

### 6.1.5 Worker: Subscribe to a process community

After the coordinator has created a process community, it is visible to any connected peer. To receive the communication concerning the `AirlineCommunity` the worker peer subscribes to the community.

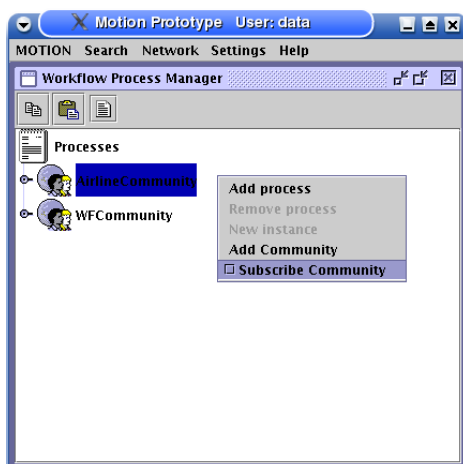


Figure 6.6: Subscribe a process community

### 6.1.6 Worker: Download process description

In order to take part in the airline reservation process, the worker peer has to receive the description of the process. This can be done using the *Workflow User Manager*. The desired process `airline` is available in the `AirlineCommunity` in the subtree of the coordinator user, named `admin`. After having downloaded the description file, it will be added to the local repository and is visible in the *Workflow Process Manager*.

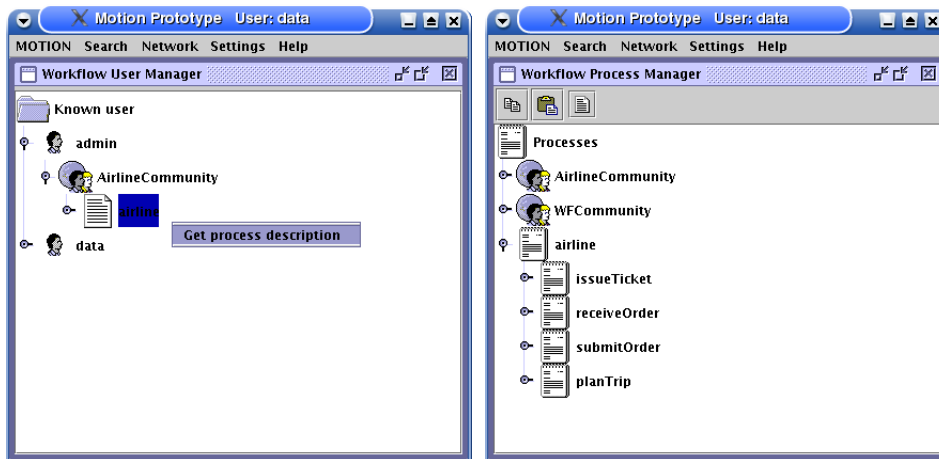


Figure 6.7: Download process description

### 6.1.7 Worker: Provide a task

To offer a task for execution, the worker has to use the *Workflow Process Manager* interface. We want to provide the task `planTrip` to the `AirlineCommunity`. After highlighting the appropriate task and using the *Copy* button we select the community it should be provided in and use the *Paste* button to proceed. Now the user is asked to specify the Quality of Service attributes. On confirmation the task is published to the community with the specified attributes and visible to the other peers.

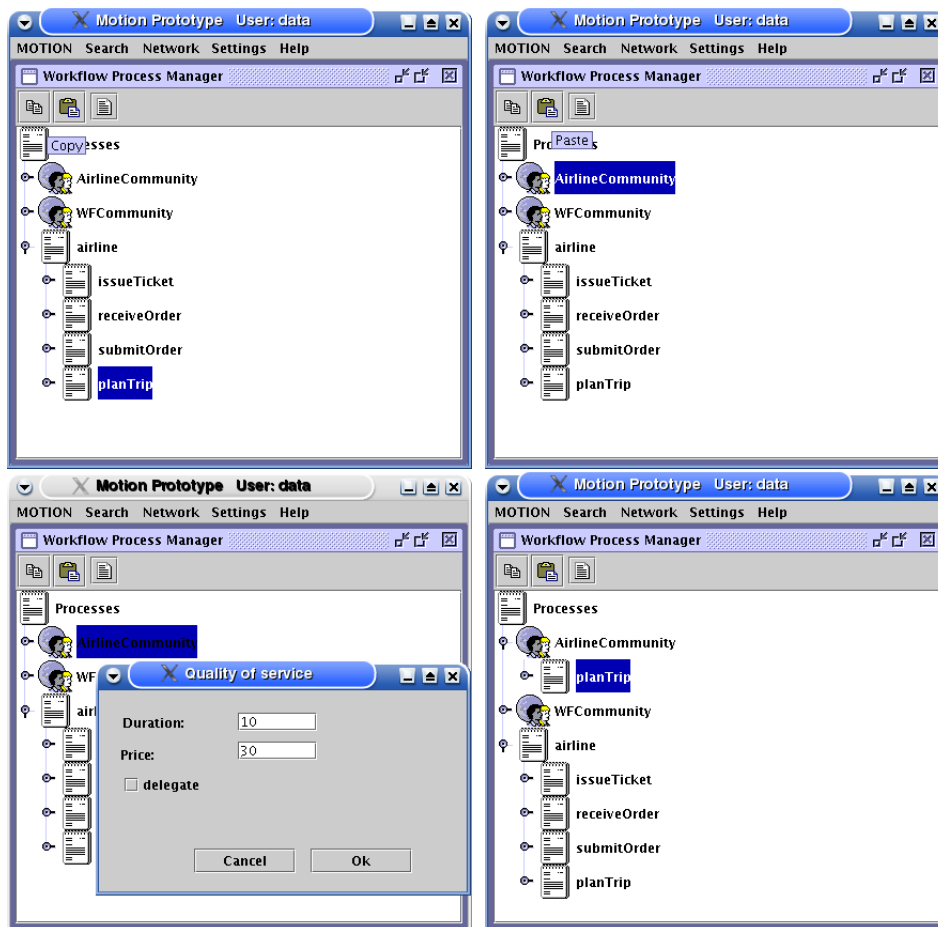


Figure 6.8: Provide a task

## 6.2 Task negotiation

In this section we will show how the coordinator looks for task providers he requires for the execution of the `airline` instance. It shows the communication with the worker peer which takes place until a task assignment is achieved. In figure 6.9 we have an overview of the steps performed.

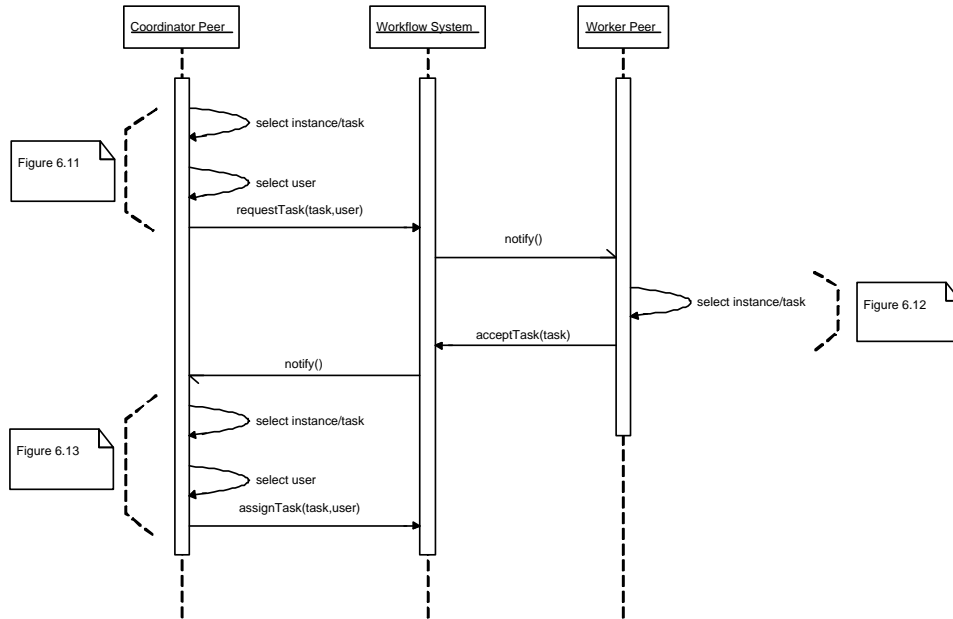


Figure 6.9: Task Negotiation overview

### 6.2.1 Coordinator: Request a task

After having created an instance the coordinator can proceed to the *Instance Overview*. This window gives an overview of all instances the peer is involved, both acting as coordinator or as worker peer. In our example the only instance shown is the previously created instance of the flight reservation scenario. To view the details and coordinate the tasks, the user can choose *Manage Instance* from the context menu.

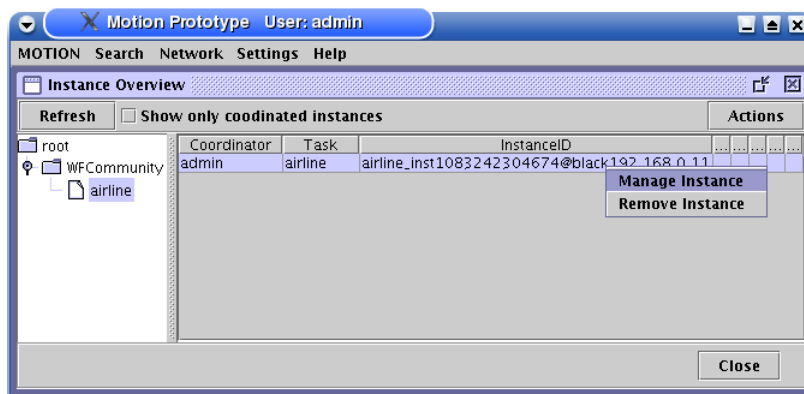


Figure 6.10: Instance coordination

The *Instance Management* window gives a detailed view of the tasks of the process. Choosing a specific task in the tree lists all peers which provide this task or which are already in negotiation with the coordinator. In our scenario we have one peer (*data*) which provides the task *planTrip* with the Quality of Service attributes listed. The coordinator can now invite the peer to execute the task with sending a *Request* message.

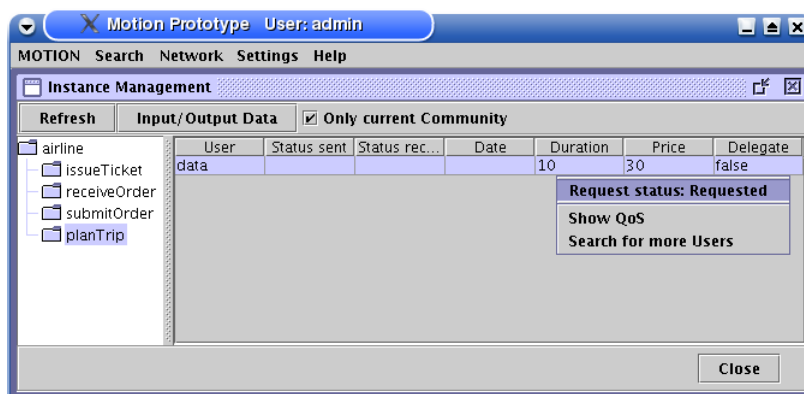


Figure 6.11: Request task

### 6.2.2 Worker: Accept a task

The worker can use the *Instance overview* window to negotiate with the coordinator. He has to choose the task `planTrip` to see all instances he is involved in and notices that the coordinator has sent a `Request` notification. He can decide whether to accept or refuse the execution with replying the appropriate status message. In this case the worker proceeds with sending an `Accept` message.

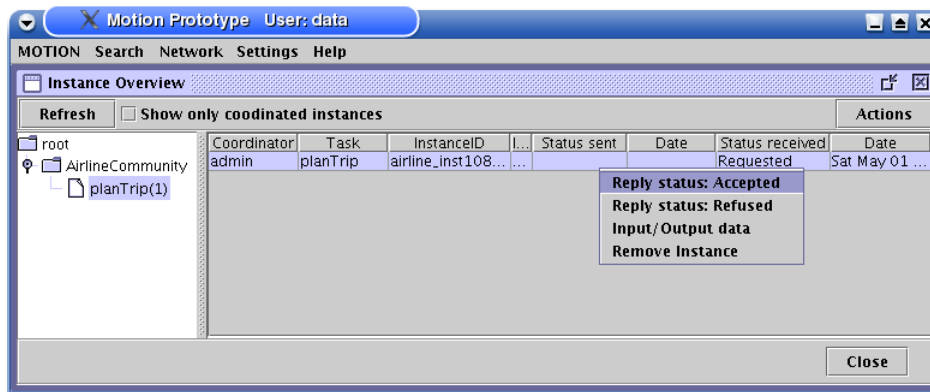


Figure 6.12: Accept task

### 6.2.3 Coordinator: Assign a task to a worker peer

The coordinator can see the latest status of the negotiation in the *Instance Management* window. To assign the task, which has been accepted by the peer, he sends another message indicating the new status *Accepted*.

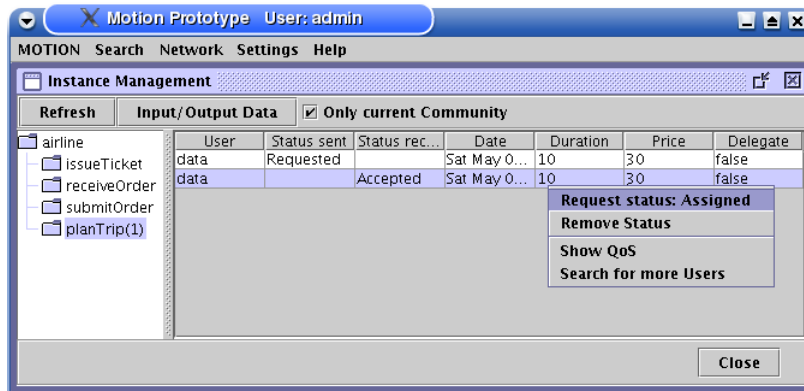


Figure 6.13: Assign task

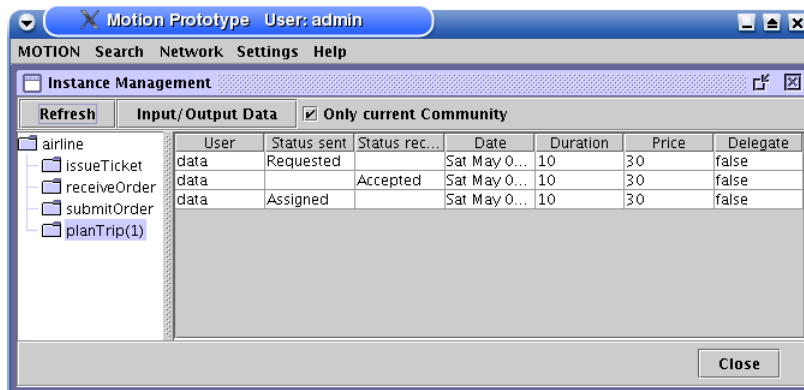


Figure 6.14: Task assignment complete

### 6.3 Instance execution without input data

In this section we show the steps which have to be performed by the worker peer to execute the task `planTrip`. According to the process description we don't require any input documents here. The handling of input data is shown in the next section.

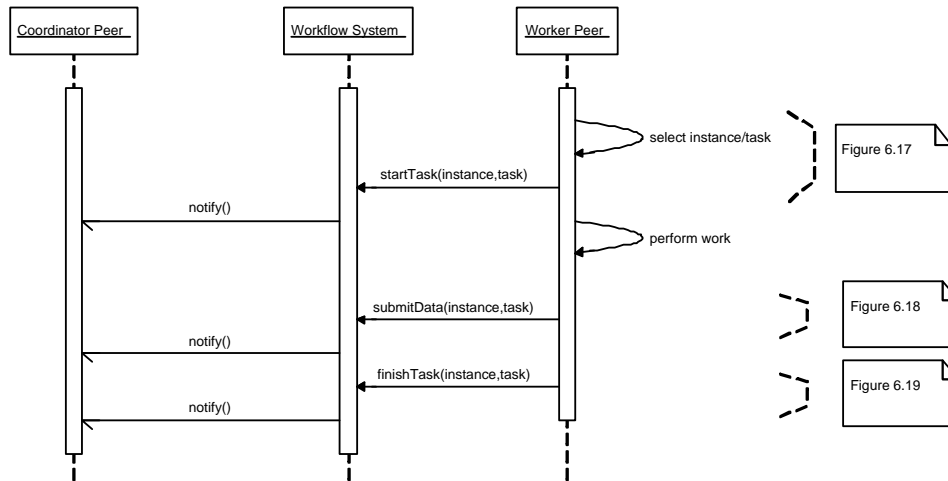


Figure 6.15: Instance execution overview 1

### 6.3.1 Worker: Start task

The worker is notified in the *Instance Overview* window about the task assignment. He can now open the *Input/Output data* dialog to have a look whether `planTrip` requires any input documents to start. As the task doesn't need any input data we have an empty list and it can be started immediately with the button *Start*. The coordinator is informed about the new status with an appropriate message.

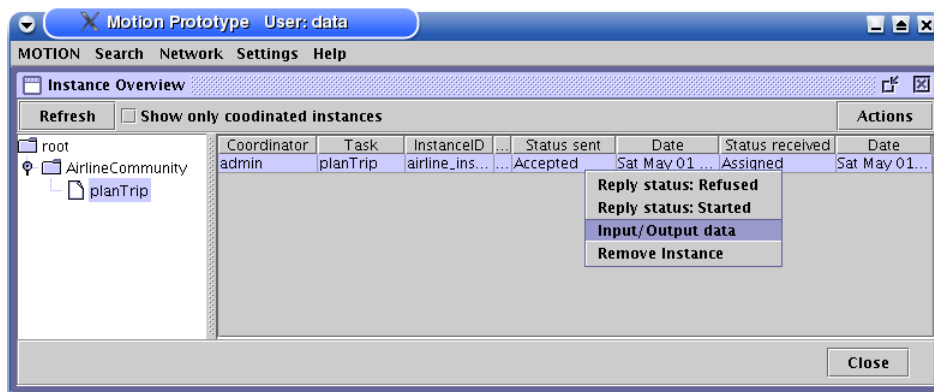


Figure 6.16: Open the Data I/O window

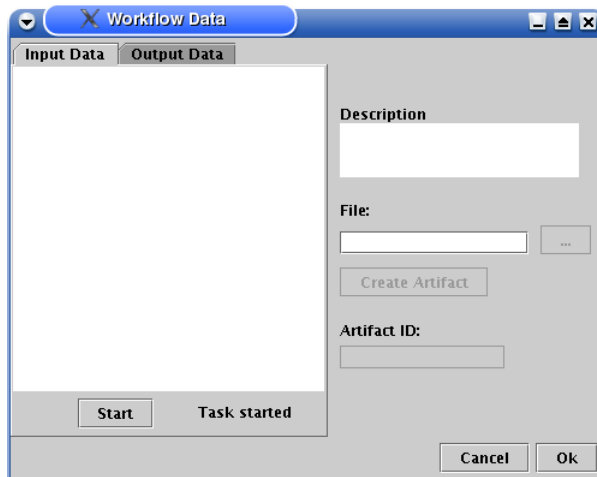


Figure 6.17: Start task

### 6.3.2 Worker: Execute and finish task

After the worker has started the task, he has any information he requires to actually perform the work. The result usually consists of one or more documents which have to be submitted to the community. To deliver the documents, if applicable, he has to use the *Input/Output data* dialog. In the output tab all required data containers are listed. For each the user has to choose a file which contains the relevant information and then create an artifact. It will be associated with the task and submitted to the instance community. In this scenario a container `locationsAndDate` has to be considered.

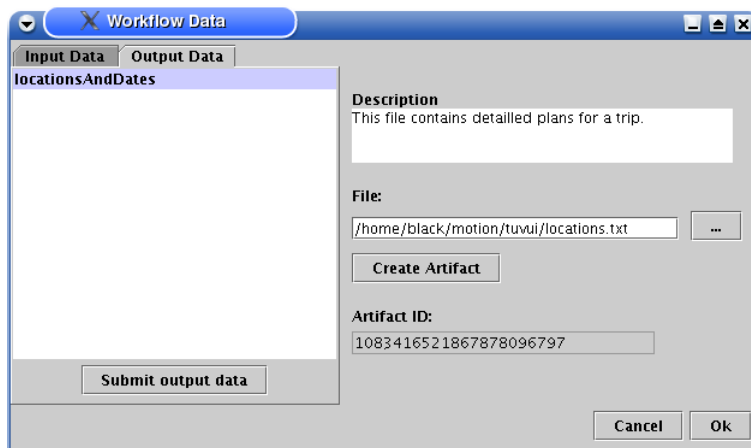


Figure 6.18: Submit task output data

Having defined all containers, the user has to click on the button *Submit output data*. This validates that all artifacts have been submitted. In positive case the task is finished and a message is sent to the coordinator indicating the new status. Otherwise the user is informed about missing documents.

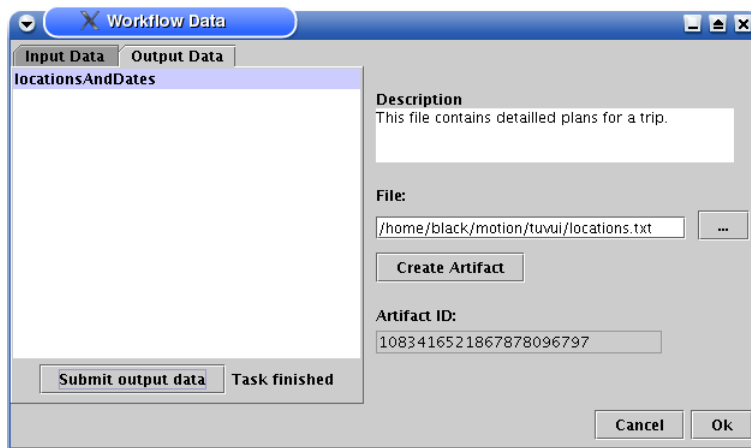


Figure 6.19: Finish task

The coordinator peer is able to track the last steps the worker has performed. In his view of the *Instance Management* window he can monitor the status transitions. After the worker has finished the task `planTrip`, the coordinator has the following information of the instance:

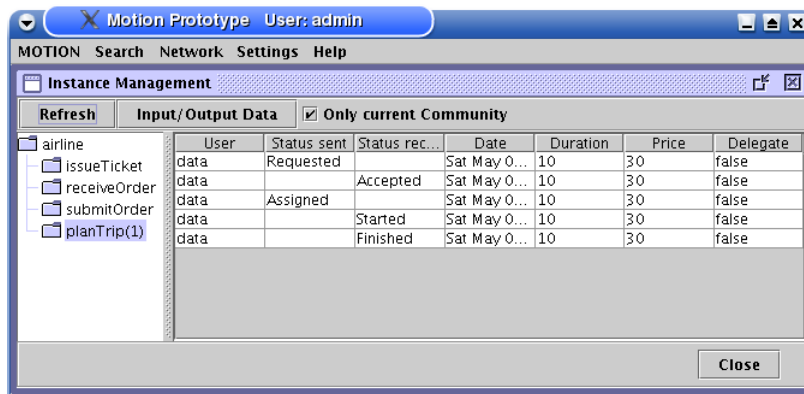


Figure 6.20: Task completed

## 6.4 Instance execution with input data

To demonstrate the execution of a task which requires input data from another peer, we assume that the following steps already have taken place:

- A peer has subscribed to the `AirlineCommunity` and has downloaded the `airline` process description
- It then has provided the task `submitOrder` to the community.
- The coordinator and the peer have successfully negotiated and the task has been assigned.

Considering these steps, the current status of the task `submitOrder` in the coordinators *Instance Management* view is shown below:

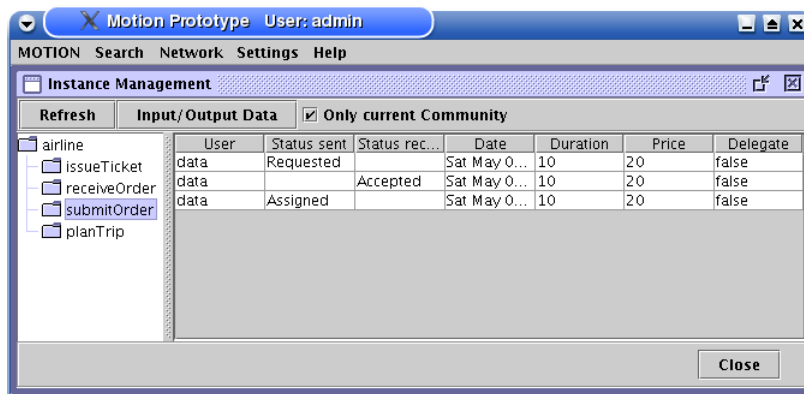


Figure 6.21: Assignment of task `submitOrder`

The worker peer now is prepared to execute the task. In the *Instance Overview* window he opens the *Input/Output data* dialog to check the required documents.

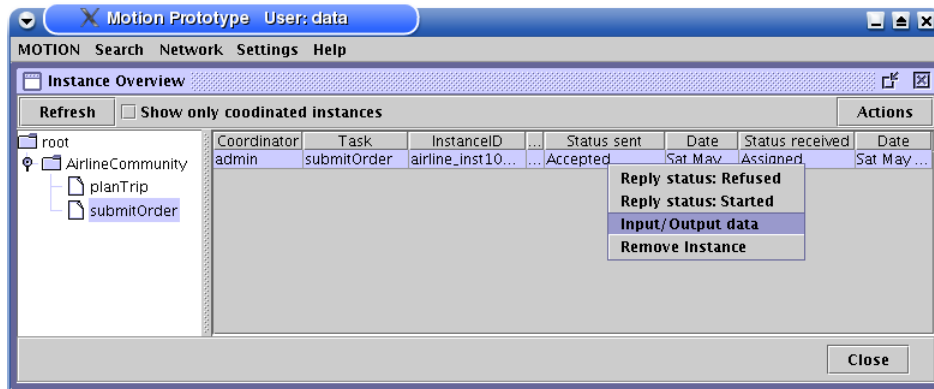


Figure 6.22: Open the Data I/O window

The *Input/Output data* dialog shows one entry in the input tab. The container `locationsAndDate` is an required input document which has to be retrieved from the instance community. With a click on the button *Get input data* the workflow system tries to download the corresponding artifact to the local fire storage. If this succeeds, the task is automatically started and a notification of the new status is sent to the coordinator. The details of the artifact are shown for the highlighted container.

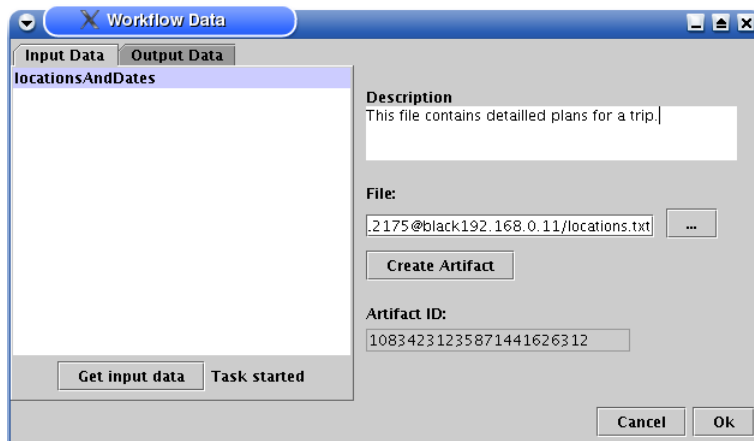


Figure 6.23: Receive input data

The *Instance Management* view of the coordinator now shows the newly received status *Started*.

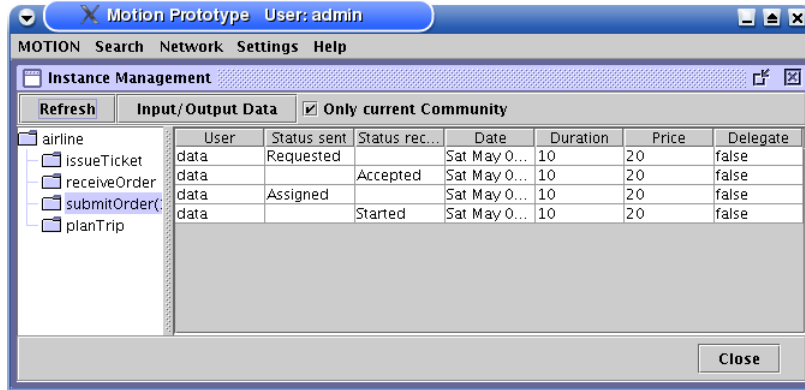


Figure 6.24: Start of task *receiveOrder*

## 6.5 Retrieve instance output documents

When all tasks are finished, the coordinator can retrieve the output documents from the instance community. He can start the *Input/Output data* from the toolbar of the *Instance Management* dialog and use it similarly to the worker peers. The difference is, that the coordinator submits input data to the instance and receives output data, whereas the worker peers receive input data and submit output data. In our example we have one output document which is to be received by the coordinator. The container `issuedTickets` is shown in the output list. With the button *Receive output data* the artifact is downloaded to the local repository, if available.

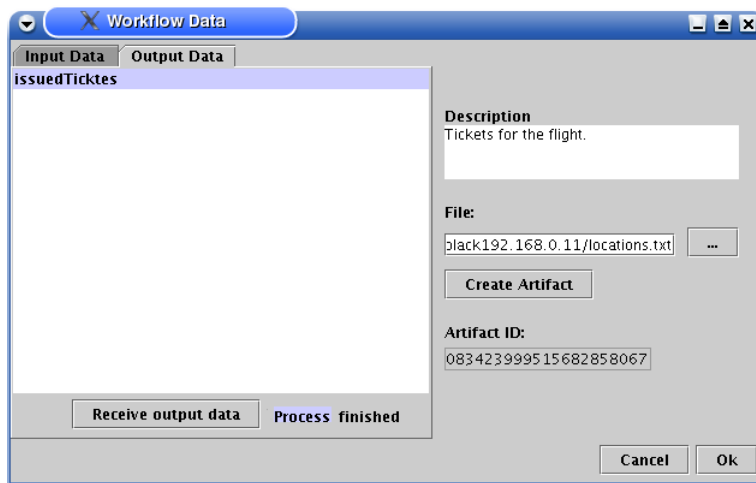


Figure 6.25: Download instance output data

## Chapter 7

# Conclusion and outlook

This thesis described a peer-to-peer workflow system targeting on the management of human workflow processes. The main elements of the architecture, i.e. the process management, the task negotiation, and the instance execution were presented and a flight reservation case study was used to demonstrate the main principles of the approach.

Peer-to-peer computing is currently expected to be a major revolution in computing, as big as the PC was in 1980s. In addition, inter-organizational workflow is becoming a main paradigm for distributed system implementation. Furthermore the rapid growth of Internet-ready mobile devices requires contribution to slim clients and limited bandwidth. Together, the three paradigms can provide the essential infrastructure for executing dynamic workflow processes over the Internet. Such workflows are adaptive (the flow of work is dynamic and is controlled by the peers) and easily scalable (i.e. by altering the number of participating peers).

Another benefit of the proposed approach is its flexibility: in a conventional static workflow system bottlenecks can occur when one of the process participants becomes unavailable. In contrast, in the peer-to-peer approach, tasks are quickly re-assigned to one of the available peers.

The discussed flight reservation case study where all peers can be assumed to be known and trusted, does not pose significant problems compared to general business workflow applications where trust, security, integrity and confidentiality are going to be of high importance. Further research is therefore required in areas such as data encryption, security and integrity of the workflow transactions. Such research includes document encryption to keep confidential information private and digital signatures to provide authenticity and integrity.

Further studies are required in providing meta-information along with process descriptions. A detailed textual and graphical description of the tasks and its dependencies is indispensable, including a definition of involved documents as regards content.

The quality of service attributes are also subject to further improvements. In addition to attributes like price or duration as introduced in this implementation, the system can supply the participants with statistically generated quality criteria. Another approach would be a rating functionality which allows process coordinators to value the quality of consumed tasks, serving others as a decision basis for peer assignment.

# List of Figures

1.1	Airline reservation . . . . .	2
2.1	The data structure managed by PeerWare. . . . .	8
2.2	Building the GVDS in PeerWare. . . . .	9
2.3	The PeerWare runtime architecture. . . . .	10
2.4	MOTION Architectural Sketch . . . . .	14
2.5	Local access to artifacts . . . . .	15
2.6	Remote access to artifacts . . . . .	15
2.7	Sketch of a workflow scenario . . . . .	16
2.8	JXTA Virtual Network. . . . .	21
2.9	The WWP architecture . . . . .	22
2.10	Healthcare service example . . . . .	24
2.11	Serviceflow model for the healthcare service example . . . . .	25
3.1	Software Architecture . . . . .	28
3.2	Community structure . . . . .	30
3.3	Create and join a community . . . . .	31
3.4	Import a new process . . . . .	32
3.5	Download process description file . . . . .	33
3.6	Provide a task . . . . .	34
3.7	Task partition of the airline reservation example . . . . .	34
3.8	Process instances . . . . .	35
3.9	Create an instances . . . . .	36
3.10	Instance lifecycle . . . . .	40
3.11	Table of peer status . . . . .	41
3.12	Task status . . . . .	42
4.1	Use case: Workflow System . . . . .	43
4.2	Task status . . . . .	44
4.3	Task Negotiation . . . . .	46
4.4	Instance Execution . . . . .	48
5.1	Class diagram . . . . .	52
5.2	BPEL example graph . . . . .	56
5.3	BPEL definition of the airline process . . . . .	59

5.4	DTD of local database . . . . .	61
5.5	Download a process description from another peer . . . . .	67
5.6	Deliver a document for an instance task . . . . .	69
5.7	Retrieve documents of an instance task . . . . .	71
6.1	Process Management overview . . . . .	73
6.2	Add a community . . . . .	74
6.3	Add process description . . . . .	75
6.4	Add a process to a community . . . . .	76
6.5	Create an instance . . . . .	77
6.6	Subscribe a process community . . . . .	77
6.7	Download process description . . . . .	78
6.8	Provide a task . . . . .	79
6.9	Task Negotiation overview . . . . .	80
6.10	Instance coordination . . . . .	81
6.11	Request task . . . . .	81
6.12	Accept task . . . . .	82
6.13	Assign task . . . . .	83
6.14	Task assignment complete . . . . .	83
6.15	Instance execution overview 1 . . . . .	84
6.16	Open the Data I/O window . . . . .	85
6.17	Start task . . . . .	85
6.18	Submit task output data . . . . .	86
6.19	Finish task . . . . .	87
6.20	Task completed . . . . .	87
6.21	Assignment of task <i>submitOrder</i> . . . . .	88
6.22	Open the Data I/O window . . . . .	89
6.23	Receive input data . . . . .	89
6.24	Start of task <i>receiveOrder</i> . . . . .	90
6.25	Download instance output data . . . . .	91

# Bibliography

- [1] John Schlesinger. What is Workflow? Business Integration Journal, March 2004, pp 40-44.  
*<http://www.bijonline.com/PDF/schlesinger%20march.pdf>*
- [2] Prof.Dr.Frank Leymann. Work Service Flow Language Specification, May 2001.  
*<http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>*
- [3] A Peer-to-Peer Middleware.  
*<http://peerware.sourceforge.net>*.
- [4] Carlo Ghezzi, Gianpaolo Cugola, and Gian Pietro Picco. A Peer-to-Peer Middleware for Mobile TeamWork.  
*[http://www.ercim.org/publication/Ercim\\_News/enw54/cugola.html](http://www.ercim.org/publication/Ercim_News/enw54/cugola.html)*.
- [5] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A Coordination Middleware Supporting Mobility of Hosts and Agents, April 17, 2003.  
*<http://www.let.polimi.it/upload/picco/papers/formalLime.pdf>*
- [6] Java Technology, Sun Microsystems, Inc.  
*<http://java.sun.com>*
- [7] Gerald Reif, Engin Kirda, Harald Gall, Gian Pietro Piccoz, Gianpaolo Cugolaz, Pascal Fenkam. A Web-based Peer-to-Peer Architecture for Collaborative Nomadic Working.  
*<http://peerware.sourceforge.net/papers/wetice2001.pdf>*.
- [8] Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, Harald Gall, Mehdi Jazayeri, Engin Kirda. Information Distribution Requirements and MOTION Architectural Principles, WP2 / Task2.1 July 30, 2000.
- [9] Microsoft Visual C#, Microsoft Corporation.  
*<http://msdn.microsoft.com/vcsharp>*
- [10] Microsoft .NET, Microsoft Corporation.  
*<http://www.microsoft.com/net>*

- [11] Mattia Monga. Supporting nomadic co-workers: an experience with a peer-to-peer configuration management tool.  
<http://homes.dico.unimi.it/~monga/lib/pbit2003.pdf>
- [12] WAP Architecture. Wireless Application Protocol, Architecture Specification, 12 July 2001.  
<http://www.openmobilealliance.org/tech/affiliates/wap/wap-210-waparch-20010712-a.pdf>
- [13] D. Hollinsworth. The Workflow Reference Model. Technical Report TC00-1003, Workflow Management Coalition, December 1994.
- [14] B. Reinwald. Workflow Management in verteilten Systemen. Teubner-Texte zur Informatik. B.G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig, 1993.
- [15] D. Georgakopoulos, M. Hornick, A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases, Kluwer Academic Publishers, September 1994.
- [16] World Wide Web consortium.  
<http://www.w3c.org>
- [17] XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). A Reformulation of HTML 4 in XML 1.0. W3C Recommendation, 26 January 2000, revised 1 August 2002.  
<http://www.w3.org/TR/2002/REC-xhtml1-20020801>
- [18] JXTA Technology: Creating Connected Communities.  
<http://www.jxta.org/project/www/docs/JXTA-Exec-Brief.pdf>, January 2004.
- [19] Sun Microsystems.  
<http://www.sun.com>
- [20] M. Corson, J. Macker, G. Cinciarone. Internet-Based Mobile Ad Hoc Networking.  
*IEEE Internet Computing*, vol. 3, no. 4, 1999
- [21] T. Dierks, C. Allen. The TLS Protocol Version 1.0. Memo RFC 2246, January 1999.  
<ftp://ftp.ietf.org/rfc/rfc2246.txt>
- [22] Georgios John Fakas, Bill Karakostas. A peer to peer (P2P) architecture for dynamic workflow management. *Information and Software Technology Journal*, 2003.
- [23] KaZaa, The KaZaa Web Site, <http://www.kazaa.com>

- [24] Zhiyong Xu, Yiming Hu. SBARC: A Supernode Based Peer-to-Peer File Sharing System.  
*[http://www.ececs.uc.edu/~oscar/papers/zyxu\\_sbarc.pdf](http://www.ececs.uc.edu/~oscar/papers/zyxu_sbarc.pdf)*
- [25] Gnutella, The Gnutella Web Site, *<http://www.gnutella.com>*
- [26] Anurag Sigla, Christopher Rohrs, Lime Wire LLC. Ultrapeers: Another Step Towards Gnutella Scalability, Version 1.0. 26 November 2002  
*[http://rhc-gnutella.sourceforge.net/src/Ultrapeers\\_1.0.html](http://rhc-gnutella.sourceforge.net/src/Ultrapeers_1.0.html)*
- [27] UDDI, The UDDI Web Site, *<http://www.uddi.org>*
- [28] Ingrid Wetzl, Ralf Klischewski. Serviceflow beyond workflow? IT support for managing inter-organizational service processes, *Information Systems* 29 (2), April 2004, pp. 127-145, 2004.
- [29] R. Klischewski, I. Wetzl, A. Bahrami. Modeling serviceflow, in: *Information Systems Technology and its Applications, Proceedings ISTA 2001*, German Informatics Society, Bonn, 2001, pp. 261-272.
- [30] W.M.P. van der Aalst, Process-oriented architectures for electronic commerce and interorganizational workflow, *Information Systems* 24 (8) (2000) 639-671.
- [31] Francisco Cubera, Yaron Goland, Johannes Klein et al. Business Process Execution Language for Web Services, August 2002.  
*<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbizspec/html/bpel1-0.asp>*
- [32] Keith Mantell. From UML to BPEL, September 9, 2003.  
*<http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel>*
- [33] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, March 15, 2001.  
*<http://www.w3.org/TR/wsdl>*
- [34] David C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, 2 May 2001.  
*<http://www.w3.org/TR/2001/REC-xmlschema-0-20010502>*
- [35] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn. XML Schema Part 1: Structures. W3C Recommendation 2 May 2001.  
*<http://www.w3.org/TR/2001/REC-xmlschema-1-20010502>*
- [36] Paul V. Biron, Ashok Malhotra. XML Schema Part 2: Datatypes. W3C Recommendation, 2 May 2001.  
*<http://www.w3.org/TR/2001/REC-xmlschema-2-20010502>*

- [37] James Clark, Steve DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999.  
*<http://www.w3.org/TR/1999/REC-xpath-19991116>*
- [38] DOM and XQL Engine, the XML Data Server - Components for Data-Intensive XML Applications. Thomas Klement, Fraunhofer's Integrated Publication and Information Systems Institute IPSI, 2002.  
*<http://www.ipi.fraunhofer.de/oasys/projects/pdom/pdome.html>*
- [39] Lauren Wood, Arnaud Le Hors et al. Document Object Model (DOM) Level 1 Specification (Second Edition) Version 1. W3C Working Draft, 29 September 2000.  
*<http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>*
- [40] Database Language SQL (Second Informal Review Draft) ISO/IEC 9075:1992. *<http://www.contrib.andrew.cmu.edu/shadow/sql/sql1992.txt>*