Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Pavel Parízek

Transactions in Peer-to-Peer Systems

Department of Software Engineering
Supervisor: **Ing. Petr Tůma, Dr.**
Study programme: **Computer science**
Study field: **Software systems**

Acknowledgements

I would like to thank my supervisor, Petr Tůma, for invaluable advice he gave me and also for his support.

I declare that I have written this master thesis myself, using only cited sources. I agree with lending and distribution of the thesis.

Prague, March 29, 2005 Pavel Parízek

Contents

Abstract

Název práce: Transakce v peer-to-peer systémech
Autor: Pavel Parízek
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Ing. Petr Tůma, Dr.
E-mail vedoucího: Petr.Tuma@mff.cuni.cz
Abstrakt: Práce se nejprve věnuje diskuzi použitelnosti transakcí v peer-to-peer systémech, s tím výsledkem, že transakce mají smysl pro některé třídy aplikací určených pro peer-to-peer prostředí, a specifikuje požadavky na protokol pro řízení průběhu transakcí, který by byl vhodný pro peer-to-peer prostředí. Dále následuje úvod do platformy JXTA, která poskytuje prostředky pro běh peer-to-peer aplikací, a do protokolu BTP, který je určen pro řízení běhu dlouhotrvajících transakcí. Poslední část textu popisuje knihovnu BTP-JXTA, která umožňuje použití transakcí v aplikacích běžících na platformě JXTA, a kterou jsem implementoval v rámci této práce. Popis knihovny se zaměřuje na architekturu a důležité implementační koncepty a také na metodiku testování, která byla použita.
Klíčová slova: transakce, BTP, peer-to-peer, JXTA

Title: Transactions in Peer-to-Peer Systems
Author: Pavel Parízek
Department: Department of software engineering
Supervisor: Ing. Petr Tůma, Dr.
Supervisor's e-mail address: Petr.Tuma@mff.cuni.cz
Abstract: The thesis begins with discussion of usability of transactions in peer-to-peer environment, with the result that transactions are useful for some kinds of peer-to-peer applications, and specifies requirements on transaction coordination protocol that is suitable for peer-to-peer environment. After that, introductions to the JXTA peer-to-peer platform and to the BTP coordination protocol follow. The last part of the thesis is devoted to the BTP-JXTA framework, which I have developed. The framework extends the JXTA platform with a transaction service that uses the BTP as its coordination protocol. The description of the framework focuses on architecture and important implementation concepts and on the methodology of testing that was used.
Keywords: transaction, BTP, peer-to-peer, JXTA

Chapter 1

Introduction

The main goal of this thesis is to design and implement a transaction service for the JXTA peer-to-peer platform. Prior to the development of the framework that will provide the transaction service, it is necessary

- to discuss usability of transactions in peer-to-peer systems,
- to specify requirements on a transaction coordination protocol that is suitable for a peer-to-peer environment,
- and to find or develop such a protocol.

The transaction service will use the protocol for coordination of transactions.

This chapter introduces peer-to-peer systems and transactions and discusses usability of transactions in peer-to-peer systems. The chapter 2 introduces the JXTA project and the chapter 3 provides introduction to the Business Transaction Protocol (BTP), version 1.0. The chapter 4 describes the BTP-JXTA framework, including sections on architecture, implementation, testing and example applications. The chapter 5 presents several well-known transaction frameworks and architectures and compares them with the BTP-JXTA framework and with the BTP, respectively. The last chapter summarizes the thesis and highlights its results.

1.1 Peer-to-Peer Systems

Peer-to-peer systems represent a new paradigm of network architecture that differs significantly from the more traditional paradigm of client-server network architecture. A client-server network consists of a reliable server and several potentially unreliable clients, which use the server.

On the other hand, a peer-to-peer system is a decentralized network of loosely-coupled autonomous peers where each pair of peers communicates directly, without a centralized server as a mediator.

The principal differences between peer-to-peer and client-server networks are:

- there are no centralized servers in a peer-to-peer network,
- peers connect directly to other peers they want to communicate with.

A peer in a peer-to-peer network has the roles of both a client and a server at the same time and thus it does not need to use a centralized server in order to communicate with other participants in the network. Peer-to-peer systems usually strive for decentralization, therefore use of centralized servers would actually be counterproductive in such an environment.

The absence of centralized servers also brings a shift in user's behavior with respect to intensive computation and data storage. The traditional client-server model promotes use of centralized servers for data storage and computation, while the peer-to-peer model expects individual peers to store data and perform computations and encourages collaboration of peers with respect to these two tasks.

Decentralization of peer-to-peer networks has the potential to bring advantages like scalability and availability, provided that the resources are replicated, and allows a peer-to-peer network to be much bigger than a client-server network. It is possible to create a peer-to-peer network consisting of millions of peers, something that would be almost impossible to handle with one centralized server. On the other hand, participants in a peer-to-peer network usually go offline and online all the time, therefore the network topology changes very rapidly and communication between peers is unreliable - these are the main disadvantages of peer-to-peer systems.

In practice, peer-to-peer systems with varying level of decentralization exist - some of them use centralized servers for tasks like user management and naming service, others use replicated servers in order to get better reliability and scalability, and there are even peer-to-peer networks that are almost completely decentralized. For example, the Napster network was quite centralized, while the Gnutella network is highly decentralized.

The JXTA Project [2] is a peer-to-peer platform that is somewhere in the middle on the scale between centralization and decentralization. It

has decentralization as one of its primary goals, but there are still some more or less centralized elements in the JXTA network.

1.2 Transactions

A transaction is, basically, a set of operations (i.e. a unit of work) that has several well-defined properties. A typical example of a transaction is a transfer of money between two bank accounts. The primary purpose of transactions is to get some form of reliability in distributed applications.

There are several transaction models, each of them defining its own set of properties that transactions must conform to. The most popular model is the *flat transaction model* that uses a set of properties well known as the *ACID* properties - because of that, flat transactions are often called ACID transactions. The letter A stands for the *Atomicity* property, the letter C stands for the *Consistency* property, the letter I stands for the *Isolation* property and the letter D stands for the *Durability* property.

Another popular transaction model is the *nested transaction model*, which differs from the flat transaction model in that it allows nesting of transactions - i.e. several transactions are executed in scope of one transaction of a higher level. There are also several models for long-lived transactions, which are usually based on the nested transaction model. More details on various transaction models, including the ACID transactions, can be found in [20].

Almost all transaction models use the *begin*, *commit* and *abort* operations. The begin operation starts a transaction, the commit operation terminates the transaction successfully and the abort operation is used if the transaction has failed. Sometimes the *confirm* term is used as an alias for the commit operation and the *cancel* or *rollback* terms as aliases for the abort operation.

Transaction systems that support ACID transactions are usually equipped with a mechanism whose task is to ensure that all participants agree on the outcome of a transaction - i.e. success or failure - in order to fulfill the Atomicity property. If the transaction succeeded then all its results are made persistent, otherwise they are cancelled, in order to fulfill the Consistency and Durability properties. This behavior is also known as the *all-or-nothing* property.

Transaction services usually support concurrent execution of ACID transactions, what means that it is necessary to use a concurrency control

mechanism to ensure that each transaction is isolated from the others, in order to achieve the Isolation property. One popular concurrency mechanism for ACID transactions is *resource locking*. It is used quite often because ACID transactions usually take a small amount of time and their scope is limited to systems that are managed by one organization, therefore it is then possible for transactions to lock necessary resources for the duration of a transaction. There is actually very rich theory behind concurrency control for transactions, but it is out of scope of this thesis to discuss even just the major ones - see [20] for more information on this topic.

Long-Lived Transactions

Long-lived transactions differ quite significantly from ACID transactions in that ACID transactions usually take only seconds to complete, while long-lived transactions can take much longer time - minutes, hours, or even days. Long-lived transactions also usually span multiple participants, which are controlled by autonomous organizations and connected by unreliable communication channels. It can be said that short-lived ACID transactions are usually used for operations upon local resources, while long-lived transactions are used for long-time business activities between autonomous participants.

From the above explanation it is clear that long-lived transactions do not use resource locking because it is not feasible to have a resource locked for the duration of a whole transaction. Nevertheless, a long-lived transaction can be composed of several ACID sub-transactions, which are local to participants controlled by one organization, and therefore able to lock resources for their limited duration.

If resource locking is not feasible for long-lived transactions, a different mechanism must be used instead - and such a mechanism are *compensating actions*. A compensating action is an operation that is used to cancel effects of a previously confirmed sub-transaction, if the higher-level transaction aborts. Compensating actions are called compensations quite often. They are usually application specific because it is not possible to automate them in a transaction framework.

Use of compensating actions instead of resource locking means that models for long-lived transactions usually relax the Isolation property, which is simply too restrictive for long-lived transactions and its use can decrease the overall performance significantly. Even the Atomicity property is relaxed in certain models of long-lived transactions.

An example of a coordination protocol for long-lived transactions that

span autonomous participants is the Business Transaction Protocol (BTP), which is described in the chapter 3.

1.3 Transactions in Peer-to-Peer Systems

Peer-to-peer systems are unstable and unreliable, as they are loosely-coupled and decentralized - they express behavior very different from client-server enterprise systems where transactions are used most often. It would, therefore, seem that transactions make no sense for a peer-to-peer environment because of decentralization and the absence of stable servers.

But, despite the unstability and unreliability of peer-to-peer networks, or perhaps just for that reason, I believe that transactions could be very useful for certain kinds of peer-to-peer applications. Even peer-to-peer applications may want to perform some operations atomically - sometimes it is necessary to send a group of messages atomically or to send the same message to more recipients, etc. Use of transactions can also provide higher reliability to peer-to-peer systems and applications, at an acceptable cost.

There is, however, one big difference between use of transactions in peer-to-peer systems and their use in client-server enterprise systems. Current enterprise systems are usually tightly-coupled, with all participants in a transaction managed by one organization, what means that it is possible to use ACID transactions and resource locking in such systems. On the other hand, transactions running in a peer-to-peer environment usually span multiple autonomous peers and behave much like long-lived transactions. In fact, long-lived transactions are targeted especially at environments like peer-to-peer networks.

Considering all the potential advantages of transactions in peer-to-peer systems, it is surprising that, as far as I know, there is no mainstream transaction framework targeted primarily at peer-to-peer systems.

In my thesis, I have developed three example applications that show possible uses of transactions in peer-to-peer applications. The applications are:

- distributed account,
- synchronization of directories,
- small platform for distributed computing.

In order to use transactions in peer-to-peer applications, it is necessary to have a transaction coordination protocol that is suitable for a peer-to-peer environment. Such a protocol must fulfill the following requirements:

- support for long-lived transactions,
- suitability for an unstable network environment,
- ability to cope with temporary unavailability of peers.

The Business Transaction Protocol (BTP) fulfills all these requirements. It is a protocol for coordination of long-lived transactions between autonomous participants in an unstable environment, what means that it is usable for peer-to-peer applications too.

Originally, I have planned to develop a new transaction coordination protocol or to modify an existing protocol for client-server systems, but I gradually found that the BTP fulfills all requirements on a transaction coordination protocol for peer-to-peer systems, therefore I decided to implement the BTP instead of developing a new protocol from scratch.

Chapter 2

The JXTA Project

This chapter introduces the JXTA Project, what is an effort to create a platform for peer-to-peer applications. The text summarizes the JXTA specification [1] and one of the overview documents [3] - both published on the JXTA Project website [2] - and highlights the principal concepts and features of the JXTA platform. As the text is based on the cited documents, it may look similar to them both in structure and content.

The whole JXTA Project is divided into several sub-projects and initiatives, which are all listed on the project's website. The core projects are the *JXTA Specification Project* [1] and the *JXTA Platform Project* [4]. The JXTA Specification Project manages the specification, which defines basic concepts and protocols, and the JXTA Platform Project is the reference implementation of the specification in the Java language. Besides these two projects, there is a lot of community projects and applications built upon the platform.

The JXTA Project was originally developed by people from Sun Microsystems, together with a few outside experts. Now it is released under the Sun Project JXTA Software License, which is based on the Apache Software License, but the development of core projects is still very much under the control of Sun Microsystems.

2.1 Basic Concepts

This section describes the basic concepts of the JXTA technology, starting with concepts related to network organization and management of information, then moving to communication-oriented concepts and finishing with the concept of the JXTA virtual network.

Network Organization

Basic elements of the JXTA network are peers, where a peer is any device that is connected to the JXTA network and that implements the JXTA specification. There are three kinds of peers distinguished by the JXTA platform - *edge peers*, *rendezvous peers* and *relay peers*.

Edge peers are normal peers that run user applications. Rendezvous peers are responsible for management of information in the network. Relay peers are related to communication between edge peers.

Peers are allowed to self-organize into peer groups, where a *peer group* is a dynamic set of peers. Peer groups are usually used to create a secure environment with a limited access or to provide custom services to a set of peers. A JXTA service is, simply said, a JXTA-enabled application. Each peer group provides a set of peer group services to its members.

The JXTA platform supports the notion of the *peer group hierarchy*, where each peer group inherits services from the parent peer group, with the possibility to override the inherited services or to use custom non-standard services in such a peer group.

The root of the visible hierarchy of peer groups is the *NetPeerGroup*, which gives each member peer the ability to exchange messages with any other peer in the network. It also provides default implementations of standard peer group services to its members. Some of the standard peer group services are conceptually defined in the JXTA specification - examples are the Pipe Service, the Discovery Service and the Membership Service, to name a few.

The actual root of the hierarchy of peer groups is the *WorldPeerGroup*, which provides peers with raw message transports, LAN connectivity and the ability to send multicast messages to other peers in a local network. The reason for the NetPeerGroup to be often considered as the root of the peer group hierarchy is that the WorldPeerGroup is hidden from applications.

Both the WorldPeerGroup and the default NetPeerGroup are automatically joined at the boot time of a peer and they both have a fixed peer group ID.

The fact that there exists something like the *default NetPeerGroup*, which has a fixed peer group ID, implies that it is possible to create a *private NetPeerGroup* with a custom peer group ID, which isolates peers in such a group from the rest of the JXTA network. Instantiation of a private NetPeerGroup is strongly recommended for the purpose of testing, in order to prevent infection of the public JXTA network.

The JXTA specification defines a *logical addressing model* that is based on location independent unique JXTA IDs, which are used for identification of resources in the network - i.e. peers, pipes, peer groups, services, etc. The fact that JXTA IDs are independent of a physical location means that a resource identified with a certain ID can use different physical addresses over time, and the JXTA platform should still work.

A separate type of a JXTA ID, together with its format, is defined in the specification for each basic JXTA resource - there are peer IDs and peer group IDs, just to name a few. It is also possible to develop a custom ID format for the purpose of a user application.

Each network resource can also have a name, besides the JXTA ID, but the names are not unique, as there is nothing like a common naming service, which would manage the names in order to ensure their uniqueness, in the JXTA network. It is nevertheless possible to use a custom naming service (DNS, LDAP) for members of a custom peer group.

Management of Information

Another important group of concepts is related to management of information in the JXTA network and to querying of those information.

Each JXTA resource has an associated *advertisement* that represents the resource to the rest of the network - we can say that an advertisement is a metadata document for the resource.

Advertisements are serialized as XML documents, with the structure defined in XML schema documents. The JXTA specification defines a special type of advertisement for each type of JXTA resource. It is also possible to define a custom type of advertisement for the purpose of a user application.

The JXTA network is also responsible for management and querying of advertisements and that is what the rendezvous peers actually do. To say it in one sentence, rendezvous peers are peers whose primary tasks are caching of advertisement indices and forwarding of queries between peers.

The relationship between edge peers and rendezvous peers is similar to the relationship between clients and servers in a typical client-server network. Each edge peer in the JXTA network should be connected to one rendezvous peer in order to be able to use the JXTA network to its full potential.

Very important is the fact that rendezvous peers cache only advertisement indices, and not the complete advertisements, in order to increase scalability of the network. An advertisement index points to the edge peer that caches the complete advertisement - when a rendezvous peer processes a query and finds an index that matches a query in its cache, it forwards the query to the edge peer that has the complete advertisement. That edge peer then sends the advertisement to the peer that issued the query. Edge peers use the Shared Resource Distributed Index (SRDI) service to push advertisement indices to rendezvous peers.

Each rendezvous peer maintains its own rendezvous peer view (RPV), what is a list of Peer IDs of several other rendezvous peers. The RPV of each rendezvous peer is regularly updated with a loosely-coupled algorithm that is based on the fact that each rendezvous peer sends a random subset of its RPV to a random set of rendezvous peers.

A loosely-consistent distributed hash table (DHT) is used to store advertisement indices at rendezvous peers. The JXTA platform uses a loosely-consistent DHT because it is more suitable for unstructured networks with unstable peers than a structured DHT and it also helps to get better scalability. More detailed description of the DHT that is used by the JXTA platform is out of the scope of this document.

Communication

The most important communication-related concepts are pipes and virtual peer endpoints, both explained here.

A pipe is a basic communication mechanism that is provided by the JXTA platform. The specification defines two types of pipes - a *point-to-point pipe* and a *propagate pipe*.

A point-to-point pipe is a unidirectional asynchronous communication channel with the input pipe end and the output pipe end. It must be supported by all implementations of the JXTA specification because the JXTA protocols rely on availability of a pipe of this kind.

A propagate pipe is a unidirectional pipe that connects one output pipe end with multiple input pipe ends.

A pipe of any kind is not bound to specific peer endpoints by default - the binding of pipe ends to peer endpoints is resolved at runtime.

The JXTA Platform project provides a few other communication mechanisms, which are built on top of unidirectional pipes - the most popular of them are bidirectional pipes, sockets and secure pipes.

The primary purpose of pipes is to allow peers to exchange messages. A JXTA message consists of a header and of an ordered set of message elements. Each message element has a type, a name and an optional namespace.

The JXTA platform supports several types of message elements - a byte array message element, a string message element and a text document message element, for example. It is possible to store an XML document into a text document message element, what implies that it is possible to implement Web Services on top of the JXTA platform - there are, in fact, projects that aim to do exactly that.

The JXTA specification defines two wire representations of messages - an XML message format and a binary message format.

Each peer uses one or more *message transports* to send and receive messages. A message transport is an abstraction of the underlying physical network transport that allows a peer to send and receive messages over that particular physical transport. All message transports that are available at a peer are encapsulated into a single *virtual peer endpoint* that allows a peer to use all available transports in a uniform way. The concept of an abstract virtual peer endpoint almost completely isolates JXTA applications from the underlying network infrastructure - the JXTA platform itself uses the most appropriate transport for sending of each message.

The JXTA specification defines the TCP/IP Transport, the HTTP Transport and the TLS Transport, together with the binding to the underlying physical network transport. It is possible to use custom transports too.

The last kind of peers, not yet explained, are relay peers. Their main purpose is to interconnect edge peers that do not have a direct physical connection. Relay peers are usable especially for NAT/firewall traversal - when a peer behind a NAT or a firewall cannot connect directly to an another peer then it must use a relay peer.

JXTA Virtual Network

The concepts that were introduced in this chapter allow a developer or a network administrator to create a virtual network on top of the existing network infrastructure. Such virtual network enables each peer to exchange messages with other peers independently of network location, firewalls and transport protocols (TCP/IP, HTTP, or other non-IP networks). Nice illustration of the structure of such a virtual network can be seen in [3].

2.2 JXTA Specification and Platform

This section gives a short overview of the JXTA specification and of the JXTA Platform project.

The JXTA specification document is divided into two parts - the first one is a conceptual overview of the JXTA technology and the second one is the actual specification that defines several protocols and a few types of advertisements and IDs. The specification itself is also divided into two parts:

- the *JXTA Core Specification* defines features that are required for all implementations,
- and the *JXTA Standard Services* defines features that are optional but strongly recommended in order to get greater interoperability with other JXTA implementations.

All JXTA protocols, which are defined in the specification, are independent of implementation, programming language, underlying operating system and network infrastructure. Protocol messages are represented as XML documents with the schema defined in the specification too. The protocols enable peers to discover each other, self-organize into peer groups and exchange messages. Some of the standard services are actually implementations of these protocols.

The JXTA Platform project is the reference implementation of the specification, which therefore implements the complete specification. In addition to features and concepts defined in the specification, it provides several extensions that make the platform significantly more usable. Examples of these extensions are sockets, bidirectional pipes and declarative configuration. The JXTA Platform project also defines the complete Java API.

Chapter 3

Business Transaction Protocol

This chapter provides a concise introduction to the Business Transaction Protocol (BTP), version 1.0, what is a coordination protocol for long-lived transactions that span autonomous participants in an unreliable environment (e.g. Internet or a peer-to-peer network). The text highlights the most important concepts of the BTP and describes principal differences between the BTP and other well-know transaction architectures and protocols, such as the JTA [6, 7] or OMG OTS [8]. Detailed description of the BTP can be found in the BTP 1.0 specification [5], which is maintained by the OASIS Business Transactions technical committee.

3.1 Multi-Level Transaction Model

The BTP supports a *multi-level transaction model*, what is another name for a *nested transaction model* with unlimited level of nesting. This feature is a must-have for any coordination protocol for long-lived transactions because limitation to no or single level of nesting would cause the protocol to be unusable for more complex transactions.

Participants in a BTP transaction form a tree-like structure. Inner nodes of the tree are called *coordinators* and leaf nodes are called *participants*. Each coordinator in the tree can have zero or more sub-coordinators and zero or more participants enrolled to it. A sub-coordinator is a coordinator that is not at the top of the tree. An application that started the transaction and communicates with the top-level coordinator is often called the *Initiator*.

Each node in the transaction tree is allowed to exchange messages only with nodes that are at most one level up or down. This rule is not explicitly stated anywhere in the BTP specification, as far as I know, but it is implicitly assumed. The parent of a node, if the node has any, is called the

superior of the node and each child of a node is called an *inferior*.

A *superior:inferior relationship* exists between each superior-inferior pair in the tree. The evolution of this relationship over time is specified by state tables that are defined in the BTP specification.

3.2 BTP Atom vs. BTP Cohesion

The BTP defines two types of long-lived transactions - *BTP Atoms* and *BTP Cohesions*.

A BTP Atom is a long-lived transaction that relaxes only the Isolation property, therefore the outcome of such a transaction is still atomic - all participants in a transaction are required to confirm in order for the transaction to succeed. Simply said, BTP Atoms preserve the all-or-nothing property. The top-level coordinator of a BTP Atom is sometimes called an *Atom Coordinator*, inner nodes are called *Sub-coordinators* and leaf nodes of the tree are called *Participants*.

On the other hand, a BTP Cohesion is a transaction that relaxes the Atomicity property in addition to the Isolation property. It therefore does not require all participants in a transaction to confirm in order for the transaction to succeed and get confirmed. Each coordinator in the transaction tree of a BTP Cohesion must discuss the outcome with the associated application, because only the application knows the set of inferiors that must confirm for the success of the sub-transaction that is managed by the coordinator. The set of inferiors that are required to confirm is called the *confirm-set*. The top-level coordinator of a BTP Cohesion is sometimes called a *Cohesion Composer*, inner nodes are called *Sub-composers* and leaf nodes in the tree are called *Participants*.

3.3 Transaction Completion

The BTP uses a slightly modified *two-phase commit protocol* for the termination of a transaction. The phases are sometimes called the decision phase and the termination phase. The *decision phase* uses the decision protocol to achieve an agreement on the outcome of a transaction and the *termination phase* then uses the termination protocol to propagate the outcome to all participants. Both phases usually meld together to a certain degree.

The termination of a BTP Cohesion is special in that each Composer in the tree must consult the confirm-set with the associated application,

while the confirm-set for a BTP Atom is always equal to set of all inferiors of a coordinator.

The BTP also supports autonomous confirm and cancel of participants, what means that participants in a BTP transaction are allowed to confirm or cancel autonomously, with the risk that it could lead to an undefined state, also known as a contradiction or a hazard.

3.4 Abstract BTP Messages

The BTP specification defines three groups of abstract BTP messages - messages used only for control relationships, messages used for outcome relationships and messages common for both types of relationships. Messages that belong to the first group are used to start or terminate a transaction - examples are the `BEGIN` and `CONFIRM_TRANSACTION` messages. Messages that belong to the second group are used to achieve an agreement on the outcome of a transaction - examples are the `PREPARE` and `CANCELLED` messages. The last group contains the `STATUS` and `FAULT` messages, to name a few.

Chapter 4

The BTP-JXTA Framework

This chapter describes the BTP-JXTA framework, which provides a transaction service for the JXTA platform, using the BTP, version 1.0, as a coordination protocol for transactions.

The chapter is divided into several sections. First, the design goals and principles for the framework are specified, then the architecture and important implementation concepts are described. The last two sections are devoted to testing and examples, in that order.

More details of topics that are related to this chapter can be found in the Programmer Guide [21], in the User Guide [22], or in appendices to this thesis. The guides have some parts that are very similar to the corresponding parts of the thesis, especially if they discuss the same feature or concept, because the thesis and the guides have different purposes and audiences - and some information must be provided to readers of any of these documents, no matter whether they prefer to read the thesis or one of the guides.

4.1 Design Goals and Principles

The design goals and principles of the BTP-JXTA framework are

- practical usability of the framework,
- lightweight architecture,
- easy learning curve,
- mapping of concepts from the BTP specification,
- and object-oriented design.

More detailed discussion of each element of the list follows.

Practical Usability of the Framework

Primary purpose of the BTP-JXTA framework is to be a transaction service for the JXTA platform, which uses the BTP as a transaction coordination protocol. My primary goal, therefore, was to develop such a transaction service that is also simple, efficient and actually usable by real-world applications.

Lightweight Architecture

I have decided to make the framework as much lightweight as possible and I have also attempted to minimize the complexity of implementation and to reduce the number of dependencies on third-party libraries.

The main motivation for lightweightness of architecture is to allow to deploy the BTP transaction service, which is provided by the framework, as a part of an application, i.e. in the same instance of the Java Virtual Machine (JVM). No separate servers or processes should be necessary in order to use the BTP service.

As for the effort to minimize the complexity and number of dependencies, I have tried to adhere to the principle that says applications and libraries should be modular, simple and should have only one purpose, in order to reduce their complexity. This principle is clearly in contrast with the practice of making applications and frameworks quite complex, dependent on a lot of third-party libraries and with a lot of unnecessary features - this applies especially to some modern frameworks written in the Java language and to a lot of specifications that are related to those frameworks.

Easy Learning Curve

Important characteristic of any framework is the learning curve. I have made an effort to design the public interfaces of the BTP-JXTA framework with simplicity in mind, focusing also on compatibility with corresponding interfaces in well-known transaction services and frameworks like the JTA [6] or OMG OTS [8], in order to make transition to the BTP-JXTA framework as easy as possible for application developers. I have also created the communication-oriented interfaces compatible with the JXTA Platform API.

Mapping of Concepts from the BTP Specification

The design and implementation of the BTP-JXTA framework is to a great extent influenced by the BTP specification. For example, the `BTPElement`, `Participant` and `ApplicationElement` interfaces correspond to abstract components that are described in the BTP specification. Also the implementation of the BTP state tables and abstract BTP messages is an almost direct mapping of the specification to the source code.

Object-Oriented Design

I have attempted to design the framework with principles of good object-oriented design in mind - for example, the *Separation of Interface and Implementation* principle is applied quite extensively. I have also used several well-known design patterns, such as the *Abstract Factory* pattern [19], during the process of development.

Clean object-oriented design of the framework results in better extensibility and easier maintenance of the source code and, what is more important, also promotes the usability of the framework in real-world applications.

4.2 Architecture

This section provides high-level description of architecture of the framework and discusses the most important public interfaces and classes that are provided by the framework.

The UML class diagram, displayed on the figure 4.1, presents the core architecture of the framework in the form of associations between interfaces. Each architectural element presented on the diagram is described below.

BTPService

```
BTPElement getBTPElement();
BTPPipeService getBTPPipeService();
```

The `BTPService` interface represents a primary access point to the BTP-JXTA framework for applications. It serves as a factory for instances of the `BTPElement` interface - it is similar to the `TransactionFactory` interface, which is provided by the OMG OTS framework, in this respect.

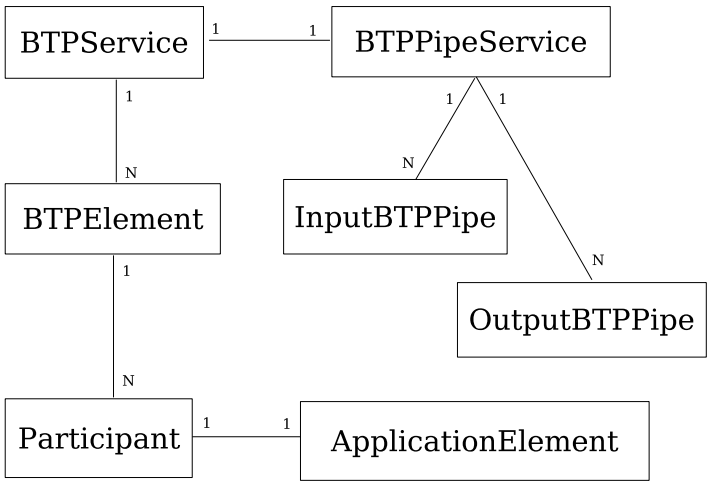

Figure 4.1: Core Architecture of the BTP-JXTA Framework

The implementation class of this interface also works as a wrapper for the JXTA Platform API, in order to allow other classes in the framework to focus on the BTP-related behavior.

An instance of the `BTPService` interface must be made available to each application that wants to use the BTP-JXTA framework for coordination of transactions. The framework was designed in a way that recommends the `BTPService` object to be deployed as a part of an application itself, although it is possible to use some remoting technology to access the `BTPService` object if it runs in a different JVM instance. It is nevertheless required to deploy the `BTPService` object at the same peer as applications that are going to use it, because each instance of the `BTPService` interface has a name that should be equal to the name of the underlying JXTA peer. Principal advantage of such a setup is permanent availability of the transaction service to applications, even if the communication channel that leads to the rest of the JXTA network is broken - and that could happen quite often in a peer-to-peer network.

An alternative approach would be to allow the `BTPService` object to exist in a different JVM instance than the application, or at a different peer. However, such an approach has several disadvantages that are based on

- unreliability of peer-to-peer networks,

- and complexity of exchange of objects between different JVM instances.

Were it the case that the `BTPService` object and the application run at different peers, it could happen that the communication channel between the application and the `BTPService` object breaks in the middle of a transaction. A solution for the problem of broken channel is migration of transaction context between instances of the transaction service, but the current version of the BTP specification does not say anything about this feature, except to note that it will be discussed in future releases.

Management of a Transaction

Management of transactions is the most important task any transaction framework should allow applications to perform. The BTP-JXTA framework provides three interfaces just for this purpose - namely the `BTPElement`, `ApplicationElement` and `Participant` interfaces. They provide means for control of application's participation in a transaction and for management of a transaction.

BTPElement

```
void begin (TransactionType type);
Participant begin (Context supCtx,
    ApplicationElement appElement);
Participant enrol(Context ctx,
    ApplicationElement appElement);
TransactionOutcome confirm(
    BTPIdentifier[] confirmSet,
    boolean reportHazard);
TransactionOutcome cancel(boolean reportHazard);
```

The `BTPElement` interface provides methods that allow an application to start and terminate a transaction and to control the transaction during its lifetime. The name of this interface is based on the *BTP Element* term, which is used in the BTP specification to refer to a component that performs BTP-related operations on behalf of an application.

Some methods of the `BTPElement` interface implement abstract BTP messages that are used in control relationships. Signatures of those methods are to a great extent determined by the BTP specification, as their arguments generally correspond to attributes of BTP messages that are im-

plemented by them. Attributes that make no sense for direct method calls are left out. The BTP specification allows abstract BTP messages that are used in control relationships to be implemented with direct method calls instead of real messages. I have decided to use direct method calls because use of real messages brings more flexibility but also more complexity. Use of direct method calls is simpler and it has only a minor disadvantage in that it ties a `BTPService` instance to the application. Fortunately, this disadvantage is not an issue for a majority of system setups.

The `BTPElement` interface must be used for both top-level transactions and sub-transactions, although some methods that are provided by the interface are not allowed to be called in case of a sub-transaction because operations that correspond to those methods make no sense for a sub-transaction - examples are the `confirm` and `cancel` methods, to name the most important of them. It would be possible to create one interface for management of top-level transactions and another interface for management of sub-transactions but that would make the API more complex without any significant advantage. I admit that the solution I have used is not as clean as it should be - it violates the *Interface Segregation Principle* [19] of object-oriented design - but it simplifies the use of the framework and reduces the learning curve.

It is also important to say that the `BTPElement` interface corresponds to specific interfaces, defined by other transaction services and frameworks - namely to the `Transaction` and `TransactionManager` interfaces from the JTA framework and to the `Terminator` and `Coordinator` interfaces, which are defined in the OMG OTS specification. All these interfaces have generally the same purpose as the `BTPElement` interface.

ApplicationElement

```
Vote prepare();
boolean confirm();
void cancel();
void cancel(Compensation[] compensations);
BTPIdentifier[] getConfirmSet();
```

The `ApplicationElement` interface allows the BTP-JXTA framework to notify applications about certain important transaction-related events and to collect responses to those events. Example of such an important event is the request to perform a decision to be prepared.

This interface has the same purpose as the `XAResource` interface, which is defined in the JTA specification, and as the `Resource`

interface, defined in the OMG OTS specification. Signature of the `ApplicationElement` interface was inspired by those other interfaces to some degree. I have only added a few methods that are necessary for my implementation of the BTP specification.

The name of this interface is based on the *Application Element* term, which is used in the BTP specification to refer to the application that uses the BTP.

Participant

```
void resign();
void allInferiorsEnrolled();
boolean autoPrepared();
boolean autoConfirm();
boolean autoCancel();
void addCompensation(Compensation compensation);
```

The `Participant` interface allows an application to control its participation in a transaction after it is enrolled to one. It especially allows an application to resign from a transaction or to perform autonomous confirm or cancel of the participant that is associated with the application. Support for such behavior is required by the BTP specification.

The methods provided by this interface implement abstract BTP messages that are used in outcome relationships and can be sent from an inferior to a superior at the initiative of the inferior, in order to support operations described in the previous paragraph.

The name of this interface is derived from the *Participant* term, which is used by the BTP specification as a name for leaf nodes in the transaction tree - i.e. nodes that do not represent coordinators.

The `Participant` interface has no mapping to any interface that is defined in the JTA or OMG OTS specifications, as those transaction frameworks do not allow resources to control their participation in transactions.

Transaction Tree

This section describes the roles of instances of the `BTPElement`, `ApplicationElement` and `Participant` interfaces in the transaction tree during the lifetime of a transaction.

Instances of the `BTPElement` interface represent inner nodes in the transaction tree (i.e. coordinators) and instances of the `Participant` and `ApplicationElement` interfaces represent leaf nodes - i.e. participants

that are responsible for application work made in the scope of a transaction. There is, however, one exception to this rule, which is described below. An instance of any of these interfaces can be associated with at most one transaction during its lifetime - this also implies the fact that it is not necessary to pass a transaction ID as an argument to methods of the `ApplicationElement` interface.

There are two key concepts that determine the structure of the transaction tree with respect to instances of the `BTPElement`, `ApplicationElement` and `Participant` interfaces. The first concept is the *difference between local and remote inferiors* and the second concept is the fact that *a sub-transaction runs at each peer* that hosts an application participating in a transaction.

A *local inferior* of a node is an inferior that is represented by instances of the `Participant` and `ApplicationElement` interfaces, which exist in the same JVM instance as the `BTPElement` object that represents the superior to the local inferior. This `BTPElement` object also manages the transaction the inferior is enrolled to. The "local" adjective reflects the fact that all these objects exist in the same instance of JVM.

A *remote inferior* of a node is represented by a `BTPElement` object that is hosted by a different peer. This also implies the second key concept, as each `BTPElement` object manages its own sub-transaction. It is then possible to enrol local inferiors, which are responsible for operations performed in the scope of a transaction, to that sub-transaction.

The decision to distinguish between local and remote inferiors was motivated by the effort to increase simplicity and efficiency. It greatly reduces the number of messages that must be exchanged between different peers, as the communication between a sub-coordinator and local inferiors that are enrolled to the sub-coordinator is implemented with method calls, not with real messages.

If the structure of a transaction tree was implemented in a different way - i.e. without the necessity to run a sub-transaction at each peer - then the implementation class of the `Participant` interface would have to be almost as complex as the implementation class of the `BTPElement` interface, because local inferiors would have to exchange messages with the remote coordinator. Such setup would therefore result in a much higher number of messages exchanged between nodes in the transaction tree.

The exception to the rule that says instances of the `Participant` and `ApplicationElement` interfaces represent leaf nodes in the transaction tree is the fact that instances of these two interfaces are created also for

each sub-transaction. The instance of the `Participant` interface, which is associated with a sub-transaction, allows the application to control the complete sub-transaction in the same way as it allows to control the participation of a local inferior in a transaction. It is not possible to use the `BTPElement` object, which manages the sub-transaction, for this purpose, because several methods of the `BTPElement` interface are allowed to be called only on a `BTPElement` object that represents the top-level coordinator, as was explained above.

Were it designed in a different way, without the exception to the rule at place, then the application would not be able to control the sub-transaction at all and it would be necessary to call a certain method on all `Participant` instances, which represent local inferiors, to do something.

The figure 4.2 displays the structure of the transaction tree for one specific transaction with respect to instances of relevant interfaces, applications and peers. It also shows the superior:inferior relationships between nodes in the tree and the direction of communication between applications and various objects that are provided by the framework.

Compensating Actions

The BTP specification does not say almost anything about distribution and execution of compensating actions - it only says that it is too much an application-dependent feature to be included in such a generic specification.

Nevertheless, the BTP-JXTA framework supports management of compensations in a very simple way. A compensating action that should be managed by the framework must be represented by an instance of the `Compensation` interface. The `Participant` interface provides the `addCompensation` method, which allows the application to supply an instance of the `Compensation` interface to the framework, and the `ApplicationElement` interface provides the `cancel` method, which passes all stored compensations back to the application if the transaction is cancelled.

It is, of course, not required from applications to use the BTP-JXTA framework for management of compensations. Applications are free to manage the compensations themselves or to use both approaches concurrently.

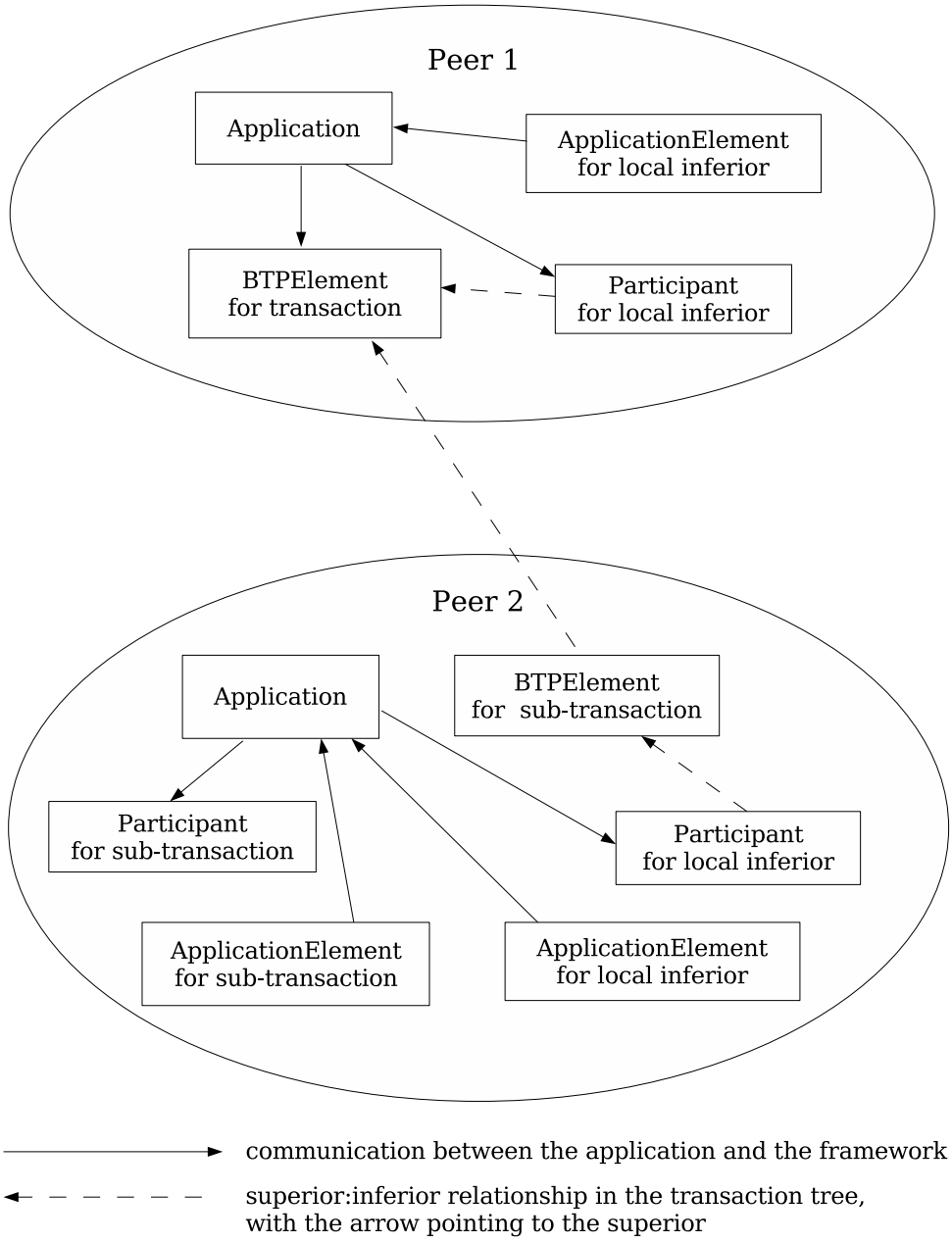

Figure 4.2: Structure of a Transaction Tree

BTP Pipes

Another important task, second only to management of a transaction, which should be performed by any framework aimed at distributed transactions, is the transfer of application messages with a transaction context attached to them, ideally in a way that is compatible with the underlying communication framework. Support for the transfer of application messages with a transaction context attached is necessary in order to give applications some information about transactions the messages belong to.

The BTP-JXTA framework also provides a mechanism for transfer of messages with a transaction context attached, which is represented by the `InputBTPPipe`, `OutputBTPPipe` and `BTPPipeService` interfaces. All three interfaces are compatible with the JXTA API to a great extent, in order to simplify the transition of existing applications built upon the JXTA technology to the BTP-JXTA framework. The *BTP pipes* term is used to refer to the interfaces listed above quite often in the text.

A BTP transaction context, represented by the `Context` interface, stores information that must be passed along with all messages exchanged in scope of the transaction, in order to allow the BTP-JXTA framework to recognize the transaction and perform all necessary operations.

The diagram, displayed on the figure 4.3, shows the classes and interfaces that make the BTP pipes, including associations between them.

The `InputBTPPipe` interface extends the `InputPipe` interface, which is provided by the JXTA Platform, and adds methods that allow an application to receive messages that have a BTP context attached to them. Messages that are received through methods inherited from the `InputPipe` interface have no BTP context attached, what means that they are not in scope of any transaction.

The `OutputBTPPipe` interface extends the `OutputPipe` interface, also provided by the JXTA Platform, and adds a method that allows an application to send messages with a BTP context attached. Messages that are sent through methods inherited from the `OutputPipe` interface are not in scope of any transaction because they have no BTP context attached, as for the `InputBTPPipe` interface.

The `BTPPipeService` interface provides methods for creating of instances of the `InputBTPPipe` and `OutputBTPPipe` interfaces - i.e. it works as a factory for instances of those interfaces. It does not extend the `PipeService` interface, which is provided by the JXTA Platform API, because methods of that interface return instances of the `InputPipe` and

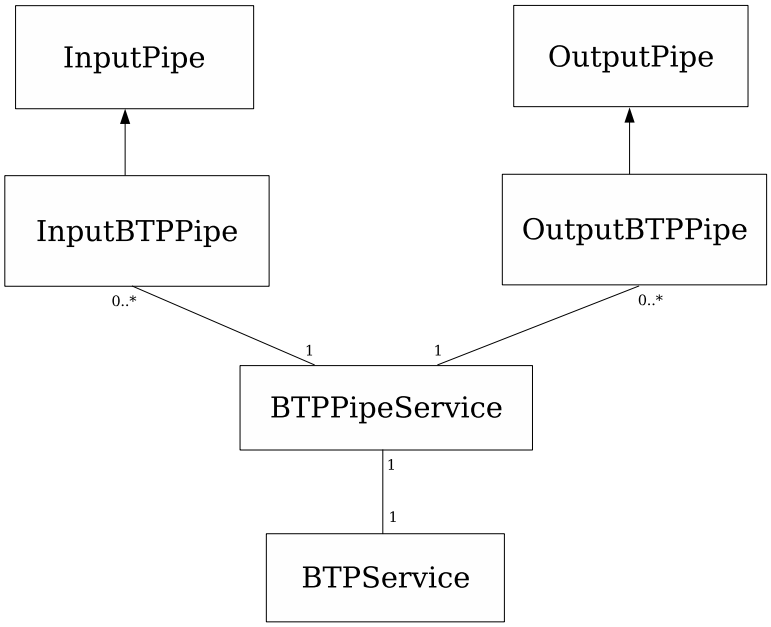

Figure 4.3: BTP Pipes

`OutputPipe` interfaces only. One instance of the `BTPPipeService` interface is associated with each `BTPService` object.

Explicit vs Implicit Transfer of Context

There are two main approaches to the transfer of a transaction context - the *explicit* and the *implicit*. These two approaches differ in that the explicit transfer requires an application to explicitly attach a context to the message, while the implicit transfer means that the framework takes care of attaching the correct transaction context to the message. The BTP-JXTA framework supports only the explicit transfer of a context for reasons that are explained below.

The fact that the BTP pipes support only the explicit transfer of a BTP context implies that the application that wants to send a message must itself pass a `Context` instance as an argument to the corresponding method of the `OutputBTPPipe` interface. When the application receives a message through some method of the `InputBTPPipe` interface, it must use the BTP-JXTA framework to extract the BTP context from the received message.

The implicit transfer of a BTP context is not supported by the BTP-JXTA framework because each `BTPElement` object must work with two `Context` instances at the same time - with the superior context and with the context for inferiors - for the reason explained in the next paragraph. An association of a `Context` object with a thread, which is a common solution for support of the implicit transfer of a context, is therefore not possible because the association would be frequently invalid.

A `BTPElement` object, which represents a node in the transaction tree, must use one `Context` instance in communication with its superior and a different `Context` instance in communication with its inferiors, instead of one `Context` instance for all cases. The reason for this is necessity to respect the rule that says each node in the transaction tree is allowed to see only one level up and one level down. A `BTPElement` object may therefore not allow its inferiors to see the `Context` instance that came from its superior.

One of possible solutions for the implicit transfer of a BTP context is to store an association of two `Context` objects with a thread but then the `OutputBTPPipe` instance would have to know whether the message is going to be sent to a superior or to an inferior and, similarly, the `InputBTPPipe` instance would have to know if the received message was sent by a superior or by an inferior. It is possible to get this knowledge in case of BTP messages but definitely not in case of application messages unless the API of the `InputBTPPipe` and the `OutputBTPPipe` interfaces is made more complex. Therefore, I have decided not to support the implicit transfer of a BTP context.

Extensions

I have also created two extensions to the core BTP-JXTA framework, which may simplify applications that use the framework for coordination of transactions on top of the JXTA platform. These extensions are described here.

FileService

The FileService extension is a wrapper for the BTP pipes that allows applications to exchange complete files very easily.

The protocol for exchange of files is designed in a way to get at least some reliability. A reply - success or error - must be received for each payload message sent by a peer, before another payload message can be

submitted. Errors detected by the sender of a file are reported with an error message and errors detected by the recipient of a file are reported with an error reply. When an expected reply is not received in the user-defined timeout, then the sender of the corresponding payload message sends an error message. This mechanism also ensures that payload messages are received in the order they were sent.

FilterInputPipe

The FilterInputPipe extension is a wrapper for the `InputBTPPipe` interface that allows applications to receive only those messages that contain a message element with a certain name.

4.3 Implementation

This section presents several important concepts and approaches that are used in the implementation of the BTP-JXTA framework.

Use of Object Factories

One of these concepts is the use of abstract factories for construction of instances of the `BTPService`, `BTPElement` and `Participant` interfaces. Each abstract factory class provides methods that create instances of the corresponding interface and the `newInstance` method, which instantiates the factory class itself. Name of the implementation class for each abstract factory can be specified with a system property; the default value is defined in the source code.

Abstract factories with customizable implementations are useful especially for testing, because it is then easy to use a custom factory class that creates instances better suited for testing - extended with additional methods, for example.

Implementation of Abstract BTP Messages

A large part of the BTP specification is devoted to abstract BTP messages that are supposed to be exchanged between participants in transactions. All abstract BTP messages, except those that are used only in control relationships, are implemented by classes that extend the `BTPMessage` class. Abstract messages that are used only in control relationships are implemented by methods of the `BTPElement` interface.

Each class that implements an abstract BTP message contains fields that are necessary to represent all attributes associated with that particular BTP message. Some of these attributes, which are common to all abstract BTP messages, are implemented directly in the `BTPMessage` class, in order to reduce the amount of source code written.

The `ContextImpl` class, which implements the `Context` interface, is not a subclass of the `BTPMessage` class because a BTP transaction context does not make a standalone message - it is sent and received only as a part of application messages.

The BTP specification defines all abstract BTP messages and their representation as XML documents for the purpose of interoperability. However, implementations of the BTP specification are free to use any representation of abstract BTP messages they want, thus the BTP-JXTA framework uses instances of the JXTA `Message` class as representation of BTP messages "on the wire", i.e. for the purpose of exchange of messages between peers. Individual attributes of a BTP message are represented as message elements. Details of mapping of abstract BTP messages to instances of JXTA `Message` class can be found in the appendix C.

Implementation of BTP Superior:Inferior Relationship

Another key concept, defined in the BTP specification and implemented by the BTP-JXTA framework, is the superior:inferior relationship and its evolution.

Implementation of the BTP superior:inferior relationship is divided into two groups of classes. Classes in the first group implement the BTP state tables, which are defined in the BTP specification, and classes in the second group implement state machines, whose purpose is to manage evolution of one instance of the superior:inferior relationship.

BTP State Tables

The BTP state tables are implemented by the `SuperiorStateTable` and `InferiorStateTable` classes, which represent the actual state tables, and by the `SuperiorState`, `InferiorState` and `Event` classes, which enumerate possible states and events.

The definition of state tables is hardwired into the `SuperiorStateTable` and `InferiorStateTable` classes, what is not a very maintainable solution, actually. I would prefer to have these two classes generated from the definition of the state tables represented as

an XML document, for example, but I have not found such a document at the website of the technical commitee that maintains the BTP specification.

State Machines

Classes that implement the BTP state tables are heavily used by the `SuperiorFSM` and `InferiorFSM` classes that implement the state machines. The "FSM" abbreviation stands for the *Finite State Machine* term.

The `BTPElementImpl` class, which implements the `BTPElement` interface, uses the state machines to manage superior:inferior relationships between participants in a transaction. The `SuperiorFSM` and `InferiorFSM` classes, also called the *state machine classes*, help to determine what to do when a certain BTP message is received or when some other BTP-related event occurs.

Addressing of BTPElement and Participant Objects

An instance of the `BTPServiceImpl` class, which implements the `BTPService` interface, must be able to uniquely identify instances of the `BTPElement` and `Participant` interfaces it manages, in order to forward each received BTP message to the expected recipient. All instances of the `BTPElement` interface, which are managed by one `BTPService` object, and instances of the `Participant` interface are identified by their *local BTP names* - i.e. strings composed of the "btpname" string and a unique number. A local BTP name must be unique only in the scope of one `BTPService` object. Each node in the transaction tree, which is represented by a `BTPElement` instance or by a `Participant` instance, is addressable by the combination of a service name, which is associated with a `BTPService` object, and a local BTP name. Instances of the `BTPAddress` class are used to represent such addresses.

Persistence

An instance of the `BTPServiceImpl` class is also responsible for persistence of instances of the `BTPElement` and `Participant` interfaces.

A `BTPService` object uses an instance of the `Persistence` interface to save the state of all instances of the `BTPElement` and `Participant` interfaces it manages. The `Persistence` interface provides methods that allow its user to save and load a byte array, what implies that the `BTPService` object must convert the state of `BTPElement` and

`Participant` objects to a byte array prior to use of the `Persistence` interface.

An application that uses the BTP-JXTA framework is free to use a custom implementation of the `Persistence` interface because instance of that interface is passed as an argument to the factory for instances of the `BTPService` interface. If no custom implementation is supplied then the default implementation, represented by the `FilePersistence` class, is used.

4.4 Testing

This section focuses on mechanisms and procedures I have used to test the functionality of the BTP-JXTA framework in order to minimize the number of undiscovered bugs in the source code.

Testing of Complex Components

Testing of software, and especially unit testing, has gained very much in popularity recently, for reasons that I would like to mention here. First of all, tools and frameworks that make the process of testing easier and more straightforward have been developed - example of such a tool is the popular `JUnit` framework [13]. Then there is the fact that the process of software development is becoming so complex that several methodologies, which are known as *agile methodologies*, have emerged and they all favor continuous unit testing of the software product from the beginning of its development to the end. Last but not least, there is also a higher demand for more reliable and secure software.

Unit testing is based on the principle that says each test case should test only one method of one class. It is hard to conform to this principle all the time, but even so the unit testing is quite usable as it helps to discover lot of bugs in the source code very early in the process. This also reduces the cost of software development in addition to the fact that the software has fewer bugs.

Another advantage of unit testing is that it enforces better object-oriented design in order to make classes and components testable. For example, components that adhere to the *Separation of Interface and Implementation* and *Dependency Injection* principles [19] are more easily testable than components that violate these principles.

However, unit testing with the `JUnit` framework, for example, is definitely not a silver bullet solution, as it is not so easy to unit test more complex components and classes that have non-trivial dependencies - and the same applies for testing of behavior that spans multiple components. It is, therefore, necessary to use an extension of basic unit testing to be able to test complex components and behavior easily, without too much effort on part of the developer.

There are actually two major approaches to unit testing of more complex components and of processes that span multiple classes.

The first approach uses *mock objects* to replace dependencies of the tested component. A mock object is something like a stub implementation of a dependency of the tested class. Few frameworks exist for making the testing with mock objects easier - for example, the EasyMock [15] and jMock [16] frameworks.

The second major approach uses the *aspect-oriented programming* to redefine certain methods in dependencies of the tested component or to trace method calls and so on.

I have decided to use the aspect-oriented programming because mock objects have one principal disadvantage from my point of view. This disadvantage is the fact that it is not so easy to create a mock object for a complex class, which has many methods - and that is the case of the `BTPElementImpl` class, which is the one I wanted to test in particular.

Introduction to Aspect-Oriented Programming

The aspect-oriented programming, usually abbreviated as AOP, is a recent paradigm of software development that extends and supplements the object-oriented programming paradigm.

In one sentence, the AOP allows a developer to implement a crosscutting concern in a more simple way than it is possible with a pure object-oriented approach. A crosscutting concern is behavior that spans multiple classes or components - logging, security or transactional behavior are popular examples of crosscutting concerns.

The following list presents the basic terminology of the aspect-oriented programming:

- a *join point* is a well-defined point in the execution of a program - e.g. method invocation or field access,

- a *pointcut* is a set of join points that exposes the execution context to the associated advice,
- an *advice* is a piece of code that is executed at each join point in a pointcut,
- an *inter-type declaration*, also called *static crosscutting feature*, changes the structure of a program; it is possible, for example, to add a field to a class or to let the class implement another interface,
- an *aspect* is a unit of code, i.e. something like a class, which encapsulates pointcuts, advices and inter-type declarations; it is possible to define normal methods, which can be used in advices, in an aspect.

The `AspectJ` framework [14] is one of the most powerful AOP frameworks and it is quite a heavyweight framework, as compared to other available frameworks. In the version I have used, it allows a developer to augment the sources at the compile time only - support for runtime modification of bytecode is going to be included in future releases, though.

Unit Testing with JUnit and AspectJ

I have used the `JUnit` framework together with the `AspectJ` framework to do unit testing of the BTP-JXTA framework.

Main motivation for use of the `AspectJ` framework, in addition to the `JUnit` framework, was the necessity to test the `BTPElementImpl` class, especially the correctness of implementation of the BTP. Use of the `JUnit` framework together with the `AspectJ` framework allowed me to perform these two tasks quite easily. The design of my testing-related classes was to a certain extent influenced by ideas from two documents I found on the web - see [18] and [17].

The following paragraphs describe the classes I have created in order to unit test the BTP-JXTA framework.

The `BTPTestCase` class extends the `TestCase` class, which is provided by the `JUnit` framework, with static methods for recording of method invocations and message submissions and receptions. The methods of the `BTPTestCase` class are static because they are called from the `BTPTestCaseRecorder` aspect. Recorded invocations of methods are stored in a list of `BTPMethodCall` objects and records of message submissions and receptions are stored in lists of `BTPMessageInfo` objects. The `BTPTestCase` class also provides several assert-methods that allow

test cases to check if a certain method was called or a certain message was sent.

The `BTPTestCaseRecorder` aspect records invocations of methods of the `BTPServiceImpl`, `BTPElementImpl` and `ParticipantImpl` classes and also messages sent or received by a `BTPService` instance.

For unit tests of the `BTPElementImpl`, `ParticipantImpl` and `BTPServiceImpl` classes I have created subclasses of those classes, which override some of the inherited methods and provide additional methods that allow test cases to prepare the state of tested objects. Moreover, subclass of the `BTPServiceImpl` class, named `BTPServiceTestImpl`, does not use the JXTA platform to send and receive messages - a test case should simulate such behavior through methods that are provided by the `BTPServiceTestImpl` class. I have also created special implementations of abstract factory classes that create instances of the above mentioned subclasses of the tested classes. These special factory classes are used only in tests.

Combined use of the `JUnit` and `AspectJ` frameworks allowed me to test my implementation of the BTP quite easily. Use of mock objects would require me to write mocks for a lot of interrelated classes, what is a problem especially in case of the `BTPElementImpl` class. With the `AspectJ` framework in place, I needed only to create subclasses of the tested classes and the special implementations of abstract factories, in addition to the actual test cases.

Functional Testing

I have also created one functional test - namely the `FileDistr` application - that runs several global transactions in order to examine various courses of a transaction. This functional test uses the `AspectJ` framework for verification of results of test cases. Implementation details can be found in the Programmer Guide [21].

4.5 Examples

This section introduces three applications that I have created as examples of use of the BTP-JXTA framework and as a proof of concept that it makes sense to use transactions in a peer-to-peer environment, at least for some classes of applications. The example applications also illustrate the usefulness of the BTP-JXTA framework for practical purposes.

I have made an attempt to create such example applications that would show three different kinds of peer-to-peer applications that can utilize transactions.

The `DirSynch` example application allows the user to synchronize content of a specific directory between two or more peers. I have choosen to create a file-exchange application because it represents a very popular class of peer-to-peer applications, a majority of which do not use transactions nowadays. One of the reasons for use of transactions by applications of this kind is to support reliable downloads in a sense of the *all-or-nothing* property.

The `Account` application provides a distributed (money) account facility to the user, what represents an another kind of peer-to-peer applications that can use transactions. This specific application illustrates use of transactions for atomic exchange of messages between several peers in order to ensure consistency of the account balance.

The last example, which I have developed, is the `DistComp` application that works as a small platform for distributed computing - and that is also a typical application for peer-to-peer systems. This example application differs from the other two in that it uses BTP Cohesions instead of BTP Atoms. It uses transactions for assignment of tasks to peers that are going to execute them.

Chapter 5

Related Work

In this chapter, I would like to present several transaction protocols and frameworks and compare them with the BTP and with the BTP-JXTA framework. More specifically, I describe the Web Services Transaction specifications, Java Transaction API, OMG Object Transaction Service and the BTP extension for the JOTM transaction service.

5.1 Web Services Transactions Specifications

The BTP is not the only coordination protocol for long-lived transactions, of course. Perhaps the most important of the other protocols and frameworks that target roughly the same environment as the BTP are Web Services Transactions that are described in the *WS-Coordination*, *WS-AtomicTransaction* and *WS-BusinessActivity* specifications [12].

The WS-Coordination specification provides a framework for definition of coordination protocols for distributed applications in the Web Services environment. The *activity* term is used as a name for general-purpose computation and communication between applications.

The WS-AtomicTransaction and WS-BusinessActivity specifications build upon the WS-Coordination specification in that they define coordination protocols that enable the participants to reach agreement on the outcome of distributed activities. Actually, the first of these two specifications provides support for short-lived atomic transactions, which have the all-or-nothing property, and the second one provides support for long-lived business activities. The BTP specification overlaps primarily with the WS-BusinessActivity specification, as the BTP is not aimed at short-lived transactions.

All three Web Services specifications are not standalone, as they are built upon other more general Web Services specifications, like SOAP and WSDL. This also implies that they use XML for representation of messages.

Use of XML for representation of messages is supported by the JXTA platform too, what implies that it would be possible to implement the Web Services Transactions specifications on top of the JXTA platform, at least in theory. However, I have decided to use the BTP as a coordination protocol partially because the Web Services specifications have several disadvantages, from my point of view. The most important of them is that they are more generic and abstract than the BTP, and they also require a couple of other specifications to be implemented first. On the other hand, the BTP is quite compact and standalone - it does not even require the use of XML, although it specifies the XML representation of abstract BTP messages for the purpose of interoperability.

5.2 Java Transaction API

The Java Transaction API (JTA) [6] is a popular API for management of transactions that is aimed at distributed enterprise applications written in the Java language.

The JTA specification defines only the API, not the actual transaction service, what implies that it is an implementation independent API. An example of a concrete implementation of the JTA is the Java Transaction Service (JTS) [7], what is a mapping of the OMG OTS to the Java environment. The JTA requires support for flat ACID transactions, and optionally for nested transactions, from the underlying transaction service.

The JTA specification defines several interfaces - the most important of them are the `TransactionManager`, `Transaction`, `UserTransaction` and `XAResource` interfaces, which are described below.

The `TransactionManager` interface represents a transaction manager that is used to coordinate distributed transactions. It corresponds to the `BTPElement` and `BTPService` interfaces, which are defined by the BTP-JXTA framework. Instance of the `TransactionManager` interface is usually used by an application server that provides a transaction service to applications hosted by it - popular example of such an application server is an EJB container. The `UserTransaction` interface is an

application-level version of the `TransactionManager` interface, as it allows an application to manage a transaction programatically.

The `Transaction` interface represents a transaction in progress. This interface also corresponds to the `BTPElement` interface, which is provided by the BTP-JXTA framework.

The `XAResource` interface encapsulates the communication with a resource (e.g. DBMS), which is performed with the X/Open XA protocol [9]. It corresponds to the `ApplicationElement` interface, which is provided by the BTP-JXTA framework.

The JTA/JTS transaction framework and the BTP-JXTA framework differ especially in the field of their target audience. The JTA is typically used for short-lived atomic transactions between several resources that are controlled by one organization, while the BTP-JXTA framework is aimed at long-lived transactions in a peer-to-peer environment. Complexity and learning curve of both APIs is roughly at the same level, in my opinion.

5.3 OMG Object Transaction Service

Another popular transaction service is the OMG Object Transaction Service (OTS) [8], which allows to use transactions in CORBA applications.

The OMG OTS specification defines only interfaces that are related to transaction management, as it builds upon several other OMG specifications - especially on the CORBA specification, to name the basic one. It uses standard CORBA mechanisms for invocation of methods on objects and for communication between participants in a transaction.

The OMG OTS specification defines many interfaces - the most important of them are the `TransactionFactory`, `Terminator`, `Coordinator` and `Resource` interfaces.

The `TransactionFactory` interface provides a method that starts a transaction and returns an object that represents the transaction. It therefore corresponds to the `BTPService` interface and partially also to the `BTPElement` interface, both provided by the BTP-JXTA framework.

The `Terminator` interface allows an application to commit or rollback a transaction and the `Coordinator` interface provides methods, which perform other operations with a transaction - registration of a resource, for example. Both these interfaces correspond to the `BTPElement` interface.

The `Resource` interface defines operations that are invoked on a resource by the transaction service. It has the same purpose as the `ApplicationElement` interface.

Primary difference between the BTP-JXTA framework and the OMG OTS is that the BTP-JXTA framework is targeted only at Java applications that use the JXTA technology for communication, while the OMG OTS is a platform and language independent transaction service, as is the case for other OMG specifications related to the CORBA technology. The OMG OTS itself is a little more complex than the BTP-JXTA framework but also more capable - it supports both the explicit and implicit transfer of a transaction context, for example. The real complexity of both BTP-JXTA framework and OMG OTS stems from the fact that they require to learn the underlying technology - JXTA or CORBA - prior to use of transactions.

5.4 BTP Extension for JOTM

The Java Open Transaction Manager (JOTM) [10] is an open source implementation of the JTA, which is hosted on the website of the ObjectWeb consortium. The important fact, with respect to this thesis, is that there is also the BTP extension for the JOTM (JOTM-BTP) [11]. The JOTM-BTP extension is an implementation of the BTP for the Web Services environment, as it uses the Axis framework that implements the SOAP as a target platform.

Chapter 6

Conclusion

This chapter summarizes the thesis and highlights important points of the text. It especially presents the results of discussion of usability of transactions in peer-to-peer systems and reviews the BTP-JXTA framework, with special focus on level of adherence to goals and principles stated prior to the development of the framework.

The discussion of possibilities of use of transactions in peer-to-peer systems results in the statement that it makes sense to use transactions for some kinds of peer-to-peer applications, despite the differencies between peer-to-peer systems and client-server systems, where transactions are used most often. Example applications for the BTP-JXTA framework, which I have developed, represent three different kinds of peer-to-peer applications that can use transactions.

The chapter, which is devoted to the BTP-JXTA framework, starts with the list of goals and principles that I have attempted to achieve and adhere to.

The level of practical usability of the framework and its learning curve are both illustrated by the example applications and discussed in the chapter 5. It is possible to say that the BTP-JXTA framework is equal with some popular transaction frameworks in terms of usability and learning curve.

The other three goals and principles are related to the architecture and design of the framework - and they were adhered to with several exceptions that are explained in the text.

I would also like to emphasize the methodology of testing I have used during the development of the framework. It is a combination of unit testing with aspect oriented programming, which allowed me to verify the functionality of the framework and the correctness of implementation of the BTP.

The BTP-JXTA framework is a working and usable transaction service for the JXTA platform, as can be seen from the example applications, but there is definitely room for improvements. I plan to make the BTP-JXTA framework, including all associated documents, publicly available, in order to allow other people to use and extend it in the future.

Bibliography

[1] JXTA Protocols Specification Project
http://spec.jxta.org

[2] Project JXTA website
http://www.jxta.org

[3] Project JXTA 2.0 Super-Peer Virtual Network, May 2003
http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf

[4] JXTA Platform Project
http://platform.jxta.org

[5] OASIS Business Transactions TC
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction

[6] Java Transaction API (JTA)
http://java.sun.com/products/jta/

[7] Java Transaction Service (JTS)
http://java.sun.com/products/jts/

[8] OMG Object Transaction Service (OTS)
http://www.omg.org/technology/documents/formal/transaction_service.htm

[9] X/Open XA Specification
http://www.opengroup.org/bookstore/catalog/c193.htm

[10] Java Open Transaction Manager
http://jotm.objectweb.org

[11] BTP Extension for Java Open Transaction Manager
http://jotm.objectweb.org/jotm-btp.html

[12] Web Services Transactions specifications
http://www-106.ibm.com/developerworks/library/specification/ws-tx/

[13] JUnit, Testing Resources for Extreme Programming
http://www.junit.org

[14] AspectJ Project
http://www.eclipse.org/aspectj/

[15] EasyMock
http://www.easymock.org

[16] jMock - A Lightweight Mock Object Library for Java
http://jmock.codehaus.org/

[17] Virtual Mock Objects using AspectJ with JUnit
http://www.xprogramming.com/xpmag/virtualMockObjects.htm

[18] Test flexibly with AspectJ and mock objects
http://www-106.ibm.com/developerworks/java/library/j-aspectj2/

[19] Martin, Robert C., "Principles and Patterns"
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF

[20] Bernstein, P.A., Hadzilacos, V., Goodman, N., "Concurrency Control and Recovery in Database Systems"

[21] Programmer Guide for the BTP-JXTA Framework

[22] User Guide for the BTP-JXTA Framework

Appendix A

Installation and Configuration of the BTP-JXTA Framework

A.1 Installation

The BTP-JXTA framework can be built and installed through invocation of the **ant** command in the directory where the `build.xml` file is located. The installation process creates the `build` directory, where all compiled class files are placed, and the `release` directory, where the results of the release build process are placed.

The `release` directory contains following files and subdirectories:

- the `btp-jxta.jar` file, which contains the BTP-JXTA framework itself,
- the `btpservice.properties` file with the default configuration of the transaction service,
- the `log4j.properties` file, which configures the `Log4j` library,
- the `pipeadvgen.sh` and `pipeadvgen.bat` utilities, which can be used to create a file with a pipe advertisement,
- the `peeradvgen.sh` and `peeradvgen.bat` utilities, which can be used to create a file with a peer advertisement,
- a subdirectory for each example application.

It is also possible to do a test build with the **ant debug** command that executes all unit tests and the `FileDistr` functional test, in addition to

building all sources. Results of the test build are placed in the `tests` directory instead of the `release` directory.

A.2 System Requirements

A few software packages and libraries are required for building and running of the BTP-JXTA framework, examples and tests successfully:

- Java runtime 1.4.2 or compatible,
- Ant 1.6.2 or compatible,
- JXTA 2.3.1 or compatible,
- Log4j 1.2.8 or compatible,
- JUnit 3.8.1 or compatible (only for the test build),
- AspectJ 1.2 or compatible (only for the test build).

The `lib` directory contains all required libraries and packages, except the Java runtime and Ant, in versions that were used during the development and testing and are known to be working together with this release of the BTP-JXTA framework. The Java runtime and Ant software packages are not distributed with the framework because they are quite common.

A.3 Configuration

The BTP-JXTA framework is configured through the `btpservice.properties` file, which must be present in the current working directory, otherwise the default configuration is used. All properties and configuration variables are described here.

The `btpservice.advertisements.directory` property defines the directory where the files with advertisements are stored. A `BTPService` instance first looks in that directory when it needs an advertisement, and only if a file with the required advertisement is not found, it tries to find the advertisement in the JXTA network.

The `btpservice.version` property specifies the version of the BTP-JXTA framework. This configuration variable should not be modified by the user.

The `btpservice.persist.filename` property defines the path to the file that is used by the `FilePersistence` class.

The `btpservice.send.initial.timeout` property defines the initial number of milliseconds between successive attempts to send a message. The initial number of milliseconds is doubled after each unsuccessful attempt.

The `btpservice.send.attempts.count` property defines the number of attempts to send a message. The failure to send a message is reported to the application only if all attempts are unsuccessful.

The `btpservice.pipe.initial.timeout` property defines the initial timeout, in milliseconds, for attempts to resolve an output pipe. The initial value of the timeout is doubled after each unsuccessful attempt.

The `btpservice.pipe.attempts.count` property defines the number of attempts to resolve an output pipe. The failure to resolve a pipe is reported to the application only if all attempts are unsuccessful.

The `btpservice.pipeadv.search.timeout` property defines the timeout for search for a pipe advertisement with the JXTA Discovery Service. Timeout value is defined in milliseconds.

The `btpservice.wait.timeout` property defines the number of seconds the `BTPService` and `BTPElement` objects wait for an important BTP-related message to arrive. For example, the wait for the `PREPARED` message is limited by the value of this property.

Appendix B

Configuration and Operation of Example Applications

Example applications are deployed into subdirectories of the `release` directory. The top-level directory for each application contains shared JAR files and a subdirectory for each preconfigured application peer. The directory for each peer contains pregenerated pipe advertisements, several properties files, and the `run.sh` and `run.bat` scripts, which start the peer. Each instance of any example application corresponds to exactly one JXTA peer and vice versa.

The most important fact related to use of the JXTA platform by example applications is that each of them uses its own private JXTA NetPeerGroup for the purpose of isolation from the public JXTA network.

All three example applications use pregenerated pipe advertisements, which are stored in XML files, thus it is not necessary to publish and discover any advertisement. Pipe names are composed of the package name and the application name. For example, the "dirsynch_peer1_pipe" string is the name of the input pipe for the peer no. 1 in the `DirSynch` application.

B.1 Configuration of the JXTA platform

All peers must have access to at least one rendezvous peer in order to work properly. Since each application uses its own private NetPeerGroup, it is safe to make a rendezvous peer from one of the application peers - actually, the peer no. 1 is configured as a rendezvous peer in all applications by default.

It is important to remember that edge peers in any example applica-

tion will not work properly (or even start) if the preconfigured rendezvous peer for the application is not running and therefore not reachable by the edge peers. If the rendezvous peer is restarted while some other edge peers are running, the edge peers may not work even after the rendezvous peer is up again.

All example applications store configuration in a properties file whose path is passed as a command-line argument. Almost all properties in the file are related to configuration of the JXTA platform. Description of individual configuration variables follows:

- the `advertisements.directory` property defines the path to the directory where the files with advertisements should be stored,
- the `peer.name` property defines the name of the peer,
- the `peer.ip.addr` property defines the peer's IP address,
- the `peer.tcp.port` property defines the peer's TCP port,
- the `peer.http.port` property defines the peer's HTTP port,
- the `peer.rendezvous` property specifies if the peer works as a rendezvous peer or not - possible values are "yes" and "no",
- the `rdvpeer.ip.addr` property defines the rendezvous peer's IP address,
- the `rdvpeer.tcp.port` property defines the rendezvous peer's TCP port,
- and the `rdvpeer.http.port` property defines the rendezvous peer's HTTP port.

B.2 Common Behavior of Example Applications

All examples write to the screen some application-specific messages and also some messages for major transaction-related events, such as *transaction begin*, *inferior enrol*, *transaction confirm*, etc. All transaction-related messages are indented and enclosed in square brackets.

The `Account` and `DistComp` applications enable a user to turn the transaction-related messages on/off with the **txon** and **txoff** commands, respectively, as there could be quite a lot of those messages. The

`DirSynch` application does not support these commands because it does not read anything from the standard input - fortunately, it does not write so much transaction-related messages to the screen as the other two example applications.

B.3 DirSynch Application

Each instance of the `DirSynch` application periodically checks the directory that is specified in the configuration. When a change is detected - a file is added, removed or updated - a notification and the complete content of the file are distributed to peers that are enumerated in the `peers.list` file. The application loads the current content of the directory at the startup and considers it to be the "original" content for the purpose of detection of changes, what means that the application does not attempt to synchronize changes that were performed when it was not running.

It is possible to cause a distribution of updates to fail (and the transaction to be cancelled) when a file without read and write access is put to the monitored directory. Such a file is then detected as an added file but the application is not able to read it and therefore the distribution fails.

The `DirSynch` application uses several application-specific configuration variables, which are:

- the `dirsynch.monitored.directory` property, which defines the path to the directory that is monitored for changes,
- the `dirsynch.uptime.minutes` property, which defines the uptime of the application in minutes,
- and the `dirsynch.monitor.interval` property, which defines the number of seconds between checks for updates to the monitored directory.

The `DirSynch` application also writes some application-specific messages to the screen. A message is displayed:

- when a new, updated or removed file is detected in the monitored directory,
- when the distribution of updated files starts,
- when the distribution is completed, successfully or with a failure,
- when a file is received, either successfully or with a failure.

B.4 Account Application

Each instance of the `Account` application accepts commands from the standard input. The **quit** command shutdowns the underlying peer immediately, the **change** command with an integer parameter changes the account balance and the **show** command displays the current account balance.

An attempt to change the account balance in a transaction succeeds if and only if all peers confirm they have received the message with a change and performed the change successfully, otherwise the transaction is cancelled. Use of transactions is necessary in order to achieve atomic changes of the account balance and to get consistent values of the account - all peers should have the same account balance, regardless of the outcome of a transaction. Whenever the account balance changes, its new value is written to a file in order to allow the peer to be restarted without breaking the consistency. The path to the file is configurable through the `account.balance.file` property.

It is possible to cause an attempt to change the account balance to fail (and the transaction to be cancelled) if the result of the change would be negative, or if at least one of the peers fails to update its local value (e.g. it is offline, etc). Nevertheless, when a peer goes offline and later it starts again, the application should work properly.

The application attempts to ensure that the account balance never falls below zero but it is not 100% reliable because of the nature of long-lived transactions. When one transaction temporarily adds a certain amount to the account balance and then another transaction substract from the balance, it is possible to get a negative value of the account balance if the first transaction aborts. This application only ensures that an individual change will not make the balance negative but it can happen that the cancellation of a transaction will make it negative.

The `Account` application writes certain application-specific messages to the screen too. A message is displayed when a local change of the account balance is performed. Other messages are displayed only at the peer that initiated the change of balance - namely the notification of result of the change and the current account balance after the change.

B.5 DistComp Application

Each instance of the `DistComp` application accepts commands from the standard input. The **quit** command shutdowns the underlying peer. The

wait command, with a task ID and a number of seconds as parameters, tells the peer to execute a task. The **stats** command displays the number of tasks that are executed at the time of invocation of the command and the total number of tasks executed during the complete session.

Each peer accepts tasks from the command-line with the **wait** command. When a peer gets a new task from the user (i.e. the peer is now the owner of that task), it tries to find a peer that will accept and execute the task - it may even be that peer itself. A peer accepts a task only if it is available at the moment the task was received. All attempts to find such a peer are encapsulated in one transaction. It can happen that the task owner would have to do several attempts to find such a peer; the maximum number of attempts is equal to three. The transaction is terminated when the task is accepted by some peer, or if the task owner runs out of attempts. Execution of the task is started when the transaction is confirmed - i.e. when the peer that accepted the task knows that the task was really assigned to it.

It is possible to cause a task to fail if the tasks are supplied to the application in such an order that results in two concurrent attempts to assign a task to one peer. This can happen, for example, when the number of tasks that are waiting to be accepted is higher than the number of application peers. If that is the case, then there will be some unsuccessful attempts to have the task accepted by a peer. If the number of unsuccessful attempts reaches the maximum then the failure to assign and execute the task is reported to the user.

If an attempt to quit an application instance is made and some tasks that are owned by the underlying peer are not completed yet, the application will only wait for the completion of those tasks before the exit - i.e. no new tasks are started during that time.

The `DistComp` application writes certain application-specific messages to the screen too. A message is displayed:

- when the actual execution of a task starts or completes,
- when the decision, if a task is accepted or not, is made,
- when the result of execution of a task is acknowledged by the owner of the task.

Appendix C

JXTA Binding for Abstract BTP Messages

This appendix describes the JXTA binding for abstract BTP messages - i.e. the way of serialization of abstract BTP messages to JXTA messages - that is used by the BTP-JXTA framework. The specification of the binding is not as formal as the definition of the XML representation of BTP messages, which can be found in the BTP specification.

The BTP-JXTA framework uses the *btp* and *btpjxta* namespaces for message elements that contain attributes of BTP messages, what means that these two namespaces are reserved for the framework. Actually, only the *btpjxta* namespace is used in this release of the BTP-JXTA framework.

Each abstract BTP message that can possibly be exchanged between two peers is implemented by a subclass of the `BTPMessage` class, and each attribute of the BTP message is represented by an attribute of that class.

The mapping of attributes of abstract BTP messages to elements of JXTA messages is defined by two tables, which can be found in the Programmer Guide [21]. I have defined the mapping for attributes of all BTP messages that are implemented by subclasses of the `BTPMessage` class, and also for some additional attributes that are used in the BTP context and in application messages. The mapping is implemented by the `BTPMessageUtils` class.

Appendix D

Level of Conformance to the BTP Specification

This appendix lists features described in the BTP specification, version 1.0, that I decided not to implement, together with motivation for the decision in each case.

Redirection

The redirection feature (and the `REDIRECT` abstract BTP message) is not implemented because the BTP-JXTA framework does not support migration of a transaction context between instances of the BTP transaction service, as a `BTPService` object with a particular service name is tied to the JXTA peer with the same name. Each received `REDIRECT` message is responded to with the `FAULT` message.

See the BTP specification, sections 5.4.4 and 7.7.16, for more details.

Timeout for the Decision to be Prepared

The `minimum inferior timeout` qualifier for the `PREPARE` message and the `inferior timeout` qualifier for the `PREPARED` message are not supported because they do not fit into the internal architecture of the BTP-JXTA framework, and they are not important enough to warrant a change to the architecture.

Time-limited decision to be prepared makes sense only when the superior and the inferior are managed by different organizations and run at

different peers. When the superior and the inferior are run by the same organization, as is usually the case of a local inferior enrolled to the superior that manages a sub-transaction, the organization can very easily ensure proper behavior of participants it controls, including timing of operations.

See the BTP specification, sections 7.10.2 and 7.10.3, for more.

Group of Related Messages

Support for groups of related messages is not implemented because it is rather complex feature and it is only an optimization of communication (i.e. an optional feature). Future versions of the BTP-JXTA framework may support this feature. An example of use of groups of messages is the one-shot optimization.

See the BTP specification, sections 5.3.2, 7.3 and 7.9, for more.

Transaction Timelimit

The transaction timelimit feature is not implemented too. The BTP specification says it is especially useful in situations when the terminator (i.e. the top-level application) fails and therefore is not able to confirm or cancel the transaction. The decider (i.e. the top-level coordinator) can cancel the transaction autonomously if the timeout expires without the terminator requesting confirm or cancel of the transaction.

This feature is not useful for my implementation of the BTP, because both the top-level application and the top-level coordinator, which is represented by a `BTPElement` object, run in the same JVM instance, what means that usually either both of them fail or none of them fails. An application should take care of doing confirm or cancel at the right time, i.e. not too long since the start of the transaction.

This feature could make some sense for sub-transactions, which should fail automatically if the top-level coordinator does not start the termination process in time. However, there is a problem in that the `BTPService` object is not able to cancel a sub-transaction in a clean way because it has no access to the `Participant` object that was returned from the `begin` method of the `BTPElement` interface, and, moreover, the decision to cancel the sub-transaction autonomously should be left to the application.

See the BTP spec, section 5.4.5, for more details on this feature.

Appendix E

Content of the Attached CD

The attached CD contains the following directories, files and documents:

- the `README` file, which contains concise description of installation and configuration of the framework and examples,
- source code for the BTP-JXTA framework, examples and tests, all in the `src` directory,
- configuration files for example applications, which are in the `main` directory,
- binary release of the BTP-JXTA framework and of the example applications, including copies of all configuration files; all of it is in the `release` directory,
- all third-party libraries required to build and run the BTP-JXTA framework and examples successfully, except the Java runtime, can be found in the `lib` directory,
- the `build.xml` and `filedistr-run.xml` files, what are build scripts used by the `Ant` tool,
- binary release of the `Ant` tool, which is in the `apache-ant-1.6.2-bin.zip` archive,
- the User Guide for the BTP-JXTA framework, whose source DocBook file and release PDF document are both in the `docs` directory,
- the Programmer Guide for the BTP-JXTA framework, whose source DocBook file and release PDF document are in the `docs` directory too,

- the `diplomka.tex` file, what is the source document for this thesis in the LaTeX format,
- the `diplomka.ps` and `diplomka.pdf` files, which both contain this document,
- the `taskspec.txt` file, which contains the original specification of this thesis,
- the `images` directory, which contains images that are used in the thesis and in both guides,
- the `BTP` directory, which contains the BTP specification and several other documents related to the BTP,
- the `JXTA` directory, which contains the JXTA specification and one overview document of the JXTA platform.