Distributed Collaborative Information Environment based on iServer

Diploma Thesis

Alexandre de Spindler
<alex@vis.ethz.ch>

Moira C. Norrie
Beat Signer

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

19th May 2005

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Contents

Abstract

The iServer architecture is a platform enabling linking of arbitrarily typed objects. It provides a metamodel defining fundamental link concepts which allow to flexibly structure and link information spaces. iServer applications have mainly been employed in a single-author-multiple-reader manner so far. The goal of this diploma thesis was to design and implement a collaborative iServer information space that allows members of a community to publish content and retrieve content published by other members. The resulting iServer web service and peer-to-peer implementations are based on a generalised interaction architecture. This architecture consists of five components each of them abstracting a particular aspect of interaction in a collaborative information system. Its main achievement is to facilitate the design and implementation of applications for distributed information sharing.

1 Introduction

Information systems not only store and give access to data but also offer facilities to semantically enrich the data. Traditionally, semantics are implied by a database schema definition which prescribes a structure all data must comply to. Structure consists of entity definitions, their attributed properties, assignment of roles and establishing of relationships [12]. Hence, relationships connecting entities make up part of the semantics contained in an information system.

A relationship consists of a connection between classes of entities. Two classes of entities are connected if the database schema provides a way of retrieving entities of one class by means of entities in the other class. The semantics of a relationship is mostly implied by a naming convention. A relationship may be defined explicitly by connecting classes of entities or implicitly in terms of a sequence of transitive connections.

Within hypertext data the `hyperlink` concept indicates another aspect of relationships. Whereas in traditional databases a relationship is defined on the class level, hyperlinks establish connections on the object level. Nevertheless, since class level relationships entail object level connections, class level relationships can also be inferred from object level connections. Thus hyperlinks can be regarded as a bottom up approach in creating a database schema and their employment clearly contributes to the semantical enrichment of the information system they are part of, i.e. the Web.

The Web can be regarded as a simple case of a database. Its entities are objects of various types such as hypertext documents, pictures, video and sound files. A link is a relationship between a source and target object. Obviously, these two observations do not lead to a semantically rich schema definition which is a well known problem with information in the Web. While the unstructuredness of Web data naturally leads to a wide variety of content, the vast amount of data and lack of semantics make it merely impossible to effectively retrieve information.

1.1 iServer

The Information Server iServer is an architecture that aims at resolving this dilemma. It combines both strengths, the structuredness of database content and the facility *to structure and link information spaces in a flexible way, while supporting a range of physical and digital media types* [14]. In fact, iServer defines a metamodel according to which objects of arbitrary types can be linked. It supports the definition of relationships on object level while preserving a database environment for structured data management. Following we give a simple and general description of iServer's main concepts.
The core of the iServer schema is the definition of an `Entity` class which comprises any object type. An iServer `Link` object consists of one or multiple source and target `Entity` objects. Thus, any type of objects can be connected by a link. The basic information units are classified as `Resources`. Additionally, a `Selector` object allows the definition of subparts of a `Resource` object to be a link source or target. `Link`, `Selector` and `Resource` are subclasses of the `Entity` class which allows a maximum flexibility in creating links. For example, a `Link` object may itself be a source or target of another link.
The iServer architecture addresses the nested link issue by introducing the concept of a `layer`. Any `Selector` object is associated with exactly one layer and overlapping layers have an explicit ordering. If a selection consists of multiple nested selectors, the selector associated to the uppermost layer will be followed. Hence it is possible to define the behaviour of nested `Selector` objects.
The iServer also supports user management that allows personalisation and content sharing based on access rights. The `Entity` class is associated with a `User` class which allows to explicitly grant or deny access to an `Entity` object for a user. The `User` class is partitioned into `Individuals` ans `Groups` for efficient user management. Additionally, each `Entity` object is associated with an `Individual` object representing its creator.

1.2 Distributed iServer

iServer databases have mainly been employed in a single-author-multiple-reader manner so far. A designated user prepares and maintains data published while consumers retrieve the available information. This one way flow of information is a widely applied publishing paradigm where information is made available within a community of anonymous users. The trustworthiness of the publisher is based on social agreement.
The advantage is that a designated publisher dedicates himself to creating and maintaining content which assures information supply. However, the creation and maintenance of a database takes much effort. Also, since the goal of a publisher must be to satisfy the needs of his entire target audience the content might not be appropriate to individual consumers.
In this work we investigate another publisher-consumer-model where consumers retrieve each other and thus act as publishers themselves. This model produces a wider variety of link structures since every member of the community is a potential publisher.
Several issues must be addressed before applying such a publisher-consumer-model. Brookshier et al. [3] name three problems arising in a community of publishers and consumers equally responsible for the available content:

- *Scrounging* is a phenomenon that can be observed in current file sharing communities.

A *leecher* is a participant which consumes without contributing. Communities consisting of too many leechers produce only little content and all the traffic is focused on few publishers which slows down the access.

- *Spamming* has come up with the popular misuse of communication channels such as e-mail and usenet. Any community with democratic publishing can be a target for unsolicited content, e.g. advertisement.
- *Malicious* content can be published for various reasons with the goal to reduce the community's worth. For example, file sharing communities have been victims of record company's attempt to publish low quality content for economic purpose.

The information supply must be assured despite the potentially vast number of publishers. It is therefore advisable to keep designated publishers which not only guarantee content quantity but also a certain level of quality. The ability of every member to publish content must be treated as an extension to the single-publisher model and not as a replacement. The goal is to produce more content in a wider variety but still to have a solid basis.
The spamming and malicious content issues can be addressed by introducing a user rating system that accounts for the lack of social agreement about the trustworthiness of individual publishers.
We implemented a system that allows members of a community to publish content and retrieve content published by other members. Instead of having one centralised database instance as in the single-publisher-model, every member runs a local instance. All members can access each other's database and the database of each member is accessible to all other members. We propose a generalised architecture enabling remote access to an extent that is set by its concrete implementation. The architecture is independent of the database it is applied for and supports arbitrary content as well as schema definitions.
This architecture regulates the interaction between members of a community. It can be used in conjunction with various technologies allowing remote communication. In this work we implemented a web service and a peer-to-peer service for iServer.

1.3 Document Structure

We introduce our interaction architecture in Chapt. 2 and present its extensions for iServer in Chapt. 3. The iServer web service and peer service implementations are presented in Chapt. 4 and 5, respectively. In Chapt. 6 we draw conclusions from our work and a user's manual is provided in Appendix A.

2

An Interaction Architecture for Distributed Information Systems

In this section we propose a formalisation of the interaction between distributed information systems. Distributed systems offer services giving access to their stored information. As opposed to commonly used databases, where a user accesses information directly, such a service is mostly consumed by another information system or program. The user accesses a remote system through a local system. Typically the two systems are of the same kind in terms of their access interface and information schema.

Our architecture defines the manner in which participants of a service exchange information. In order to simplify the understanding of the architecture, we assume a simple request response model. Regardless whether we are considering a web service or a peer-to-peer service, the moment a participant polls another participant, those two can be regarded as engaging into a client-server relationship handling a request with a response. To avoid any confusion we use the terms `local` to denote a client requesting information and `remote` for a server responding on request. Note that in a peer-to-peer service a peer does both, sending requests as well as responses.

In Fig. 2.1 we propose a general interaction schema according to which services such as web and peer-to-peer can be implemented. A database API defines methods to retrieve, manipulate, add and remove data stored in an information system. Any request to the system must either be an API method call or a higher level operation composed of multiple API method calls. The goal of a service is to offer the database functionality of a remote system to a local system.

The figure depicts a situation where a local system makes a request to a remote one. It shows all required components. We use `Data` objects to encode the request to be processed remotely and the resulting response. A `Request` object is created by a request factory which can be regarded as a local interface to the database functionality offered by the remote system. The request is handed over to a `Local Request Handler` (a) which creates a `Message` object containing it. A message wraps a `Data` object. It can generate an `XML`

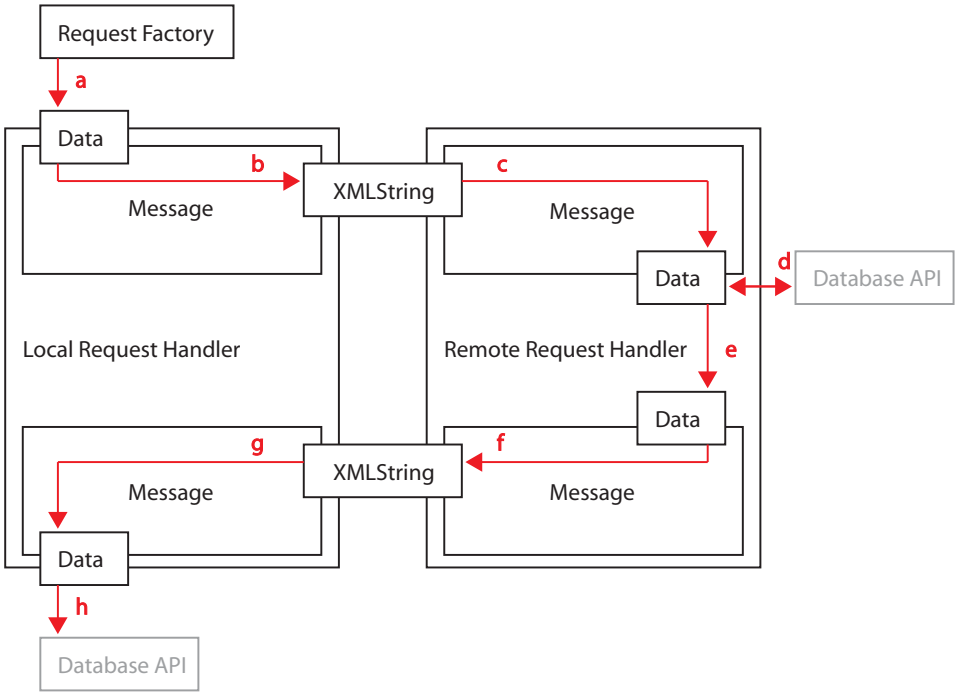

Figure 2.1: General schema for interaction between distributed information systems.

`string` from which it can be reconstructed (b). The string value is transmitted to a `Remote Request Handler` which reconstructs the message and extracts the `Request` object (c). A request processes itself by independently interacting with the API of the remote database (d) and returns a `Data` object containing the response (e). The remote handler just initiates the request processing and uses the `Response` object to create a `Message` object which is returned to the local request handler as XML string (f). The local handler reconstructs the message and extracts the response (g). The response can either be presented directly to the user or fed back to a local information system (h). In the second case the data stored in the remote system can be transparently made available to the local system.
In this general interaction schema we identify four components which participate in the interaction between the local and the remote information system: a local request factory, request handlers, data and message objects. A fifth component, which is not shown in Fig. 2.1, is the protocol schema definition according to which data objects are represented as XML strings. In sections 2.1 - 2.5 we present these components individually while in Sect. 2.6 we outline the interplay of the components and point out the advantages of implementing a service by using them.

2.1 Request Factory

A request factory constructs `Request` objects that encode a request to a remote information system. It can be regarded as the interface to a remote system. A request may represent a single database API method which is to be invoked on a remote system. It may also represent

a higher level operation consisting of multiple database API method calls. The factory contains a static method for each request that can be processed remotely. Each method returns a `Request` object encoding the particular request it represents.
A request factory can be implemented according to the functionality designed to be offered by the service. It may replicate the database API completely or partly and offer service-specific higher level operations. The set of its static methods defines the complete functionality offered by the service, since it is the only place where `Request` objects are created. A request factory must be implemented specifically for each interface that is to be offered as service, for each database API that is to be used to process the request and for each request pattern.
Figure 2.2 shows a pseudo UML diagram of the request factory. All API methods are represented with the expression in squared brackets. Note that a request factory does not necessarily replicate database API methods but may also combine multiple API method calls with one request.

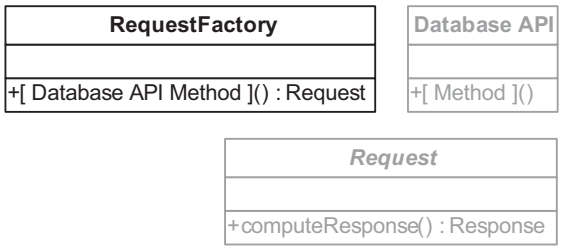

Figure 2.2: The request factory forms the interface to a remote information system.

2.2 Handling

Handlers are used to initiate and control the processing steps required to implement a service. Handlers are implemented specific to a service framework such as web service or peer-to-peer service. Figure 2.3 shows the UML diagrams of a typical local and remote request handler. A `LocalRequestHandler` object handles a `Request` object and returns a `Response` object. A `RemoteRequestHandler` handles an XML string representing a message containing the request and returns an XML string representing a message containing the response.

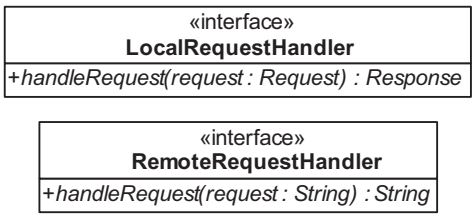

Figure 2.3: Handlers implement the service specific requirements for communication with a remote system.

The local request handler transforms the `Request` object into an XML string using a

`Message` object. The string value is transmitted to the remote request handler. Therefore the local handler must be able to contact the remote handler and to transmit a string value. Since the procedure of locating, contacting and transmitting a message to a remote system varies with the service framework used, the handlers must be implemented specifically for each framework. They encapsulate the framework specific requirements and implement interface methods that use `Data` and `String` parameter and return types. The handling component ensures that all other components are independent from the service framework.
The remote request handler uses the XML string received to reconstruct a corresponding `Message` object from which it extracts the `Request` object. Since the `Request` object processes itself, the handler only needs to initiate the processing and construct another message containing the response resulting from it. The XML string representing the response message is returned to the local remote handler where the `Message` object is reconstructed. The local handler extracts the response which can either be presented directly to the user or fed back to the local system.
Alternatively, a request response pattern can be implemented so that the response is not returned by the local request handler as described above. Instead, a `ResponseHandler` object registered with the request handler is notified whenever a response arrives. The response handler processes the response, e.g. by feeding it back into the local database. This pattern is appropriate when a request causes multiple responses or when a response to a request cannot be expected immediately.

2.3 Request and Response

The core of our interaction architecture consists of `Data` objects representing a request or response. The `Request` and `Response` objects are wrapped in a `Message` object that generates its XML string representation from which it can be reconstructed. A `Message` object generates an XML document and adds an XML element representing the `Data` object it wraps. Thus, any data to be wrapped in a message must offer methods to generate an XML element and another method from which it can be reconstructed from an element. In Fig. 2.4 we define an abstract class `Data` that declares these methods. Hence, any type of object derived from the `Data` class can be wrapped in a message. Implementations of the `Data` class use the element factory introduced in Sect. 2.5 to convert their own member objects from and to XML elements. The method `toXML(String)` takes as argument a string value which is used as the name of the generated XML element. Thus, a `Message` object is able to name the children elements of its representing root element according to the XML schema definition. The schema definitions are introduced in Sect. 2.5.
A `Message` object that is reconstructed from an XML string parses the string and extracts the XML element representing the `Data` object it contains. The `Data` object is reconstructed using Java Reflection which requires the name of the class to be constructed. Reflection is a feature in the Java programming language that allows an executing program to examine and manipulate its own classes, their fields and methods. Among other things classes can be created and methods executed based on their names and signatures only. Therefore, the XML element representing a `Data` object contains an attribute holding the class name. However, sometimes we do not need to reconstruct a `Message` object from its XML string representation, e.g. a response message in a web service. Therefore we implemented a `toNonJavaXML(String)` method not shown in Fig. 2.4 for the `Data` class which returns

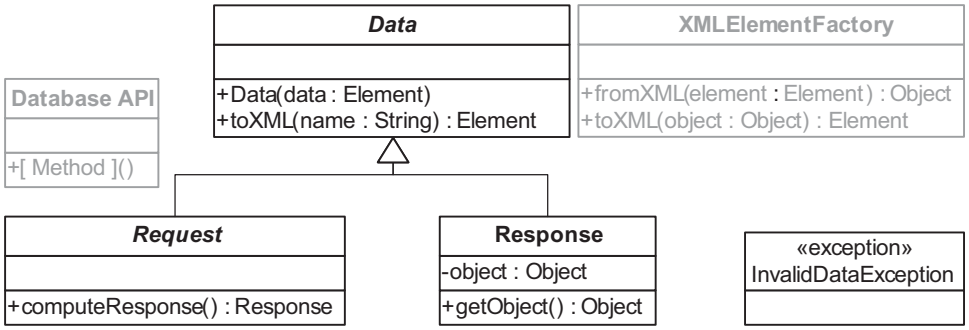

Figure 2.4: `Data` objects used to represent requests, responses and any other object with a registered marshaler. `Data` objects can be wrapped in message objects.

an XML element without the class name attribute.
The constructor of the `Data` class throws an `InvalidDataException` in case an object cannot be constructed with the XML element given as parameter. This exception declares a constructor taking another exception as argument which is considered as its cause. Typical causes are `MalformedXMLException` and `NotRegisteredException` which will be introduced shortly.
A request to a remote information system encodes the database API method calls necessary to obtain the result. In our work we have implemented a `Request` subclass that uses Java Reflection to encode and execute an iServer API method invocation. We have also implemented another set of `Request` subclasses that allow database querying based on *select* and *intersect* operations which can be nested (Appendix C). In order to be able to support arbitrary request patterns we define an abstract `Request` class that ensures the availability of a `computeResponse()` method returning a `Response` object. The handlers rely on this method only and, thus, all other components of our architecture stay independent from the request implementation.
Figure 2.5 illustrates the processing of a request. The method initiates the processing of the request which is carried out by the `Request` object interacting with the database API (a and b). The method creates and returns a `Response` object representing the result (c).
A `Response` object contains the result of a request which allows it to be wrapped in a message and be represented as and reconstructed from an XML string. The `Response` class shown in Fig. 2.4 contains a member of type `Object` where the response to a request is assigned to. The method `getResponse()` gives access to this member. Subtypes of the `Response` class can be implemented that cast the result object to a specific type inside this method. The request factory knows which type of object is returned by a particular request. Therefore, it constructs objects of a type extending the abstract `Request` class. These objects return `Response` objects of a subtype of the `Response` class. For example, if a particular request is known to return an integer value the request factory constructs an object of type `IntegerRequest` which extends the abstract `Request` class. The method `computeResponse()` in the integer request class returns an object of type `IntegerResponse` that extends the `Response` class. Its `getObject()` method returns an integer object.
Since a `Request` object interacts with the database using its API methods and since the

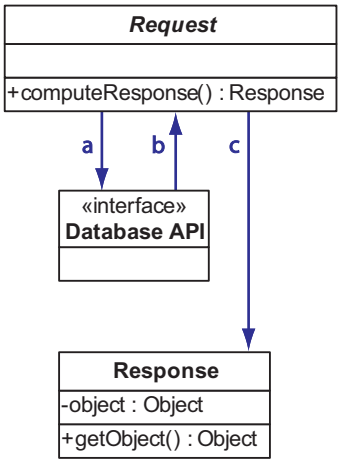

Figure 2.5: Workflow within the `computeResponse()` declared by the `Request` class.

`Response` objects contain results corresponding to the API methods' return types, they must both be implemented specifically to a database interface definition. The `toXML(String)` and `fromXML()` methods must return XML elements conforming to the protocol schema definitions presented in Sect. 2.5.
In the iServer peer service introduced in Chapt. 5 responses received are compared for rating. Therefore, the `Response` class overrides the method `equals(Object)` by delegating the equality request to its `Object` member.

2.4 Messaging

In our interaction architecture requests and responses are transmitted as XML strings. `Request` and a `Response` objects represent the request to a remote information system and the resulting response. In order to facilitate the transformation of a `Data` object to its string representation and vice versa we designed a `Message` class (as shown in Fig. 2.6). An instance of the `Message` class can be constructed with a `Data` object it must contain. The method `toXMLString()` generates its XML string representation. A `Message` object can also be instantiated from a string value using the `Message(String)` constructor. The `Message` class gives access to the data member through an accessor method. Both constructors throw an `InvalidMessageException` if the `Data` object is not valid or if the XML element does not represent a valid `Message` object. The exception declares a constructor with an `Exception` typed parameter that is regarded as the cause of the former. A `Message` object generates its XML string by first creating an XML document. Then, the XML element created by the member `Data` object is added to it. The generated string value is a representation of this XML document. The other way around, an XML document is constructed by parsing the string value. The element representing the `Data` object is passed to the `Data` class constructor and the instantiated `Data` object is assigned to the member of the `Message` object.
The constructor is invoked using Java Reflection. The `Class` object of the member `Data` object is instantiated using its name. Therefore, the XML element representing the `Data`

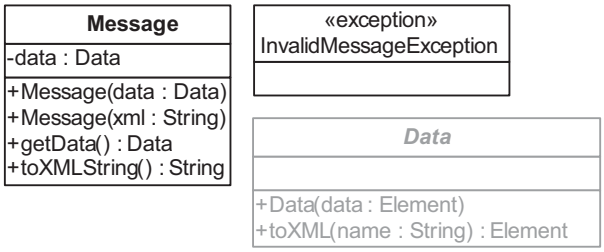

Figure 2.6: The `Message` class is used to transform `Data` objects into XML strings and vice versa.

object must have an attribute holding the class name. The `Message` class defines a method `toXMLString(boolean)` taking a boolean value controlling whether the contained `Data` objects shall contain this attribute or not. Unnecessary information transfer can be avoided if a message does not have to be reconstructible.
Figure 2.7 illustrates a request-response-interaction between a local and a remote request handler. The local handler creates a message containing the `Request` object (a) and sends its XML string representation to the remote handler (b). There the `Message` object is reconstructed from the string value (c) and the `Request` object extracted (d). The `Response` object resulting from the request processing is wrapped in a message (e) and its XML string representation returned to the local handler (f) where the message is reconstructed and the response is extracted.

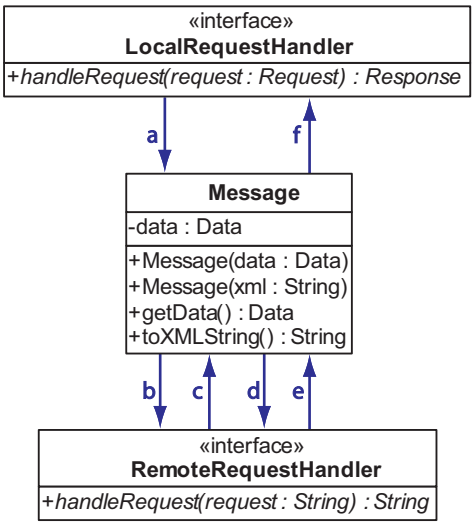

Figure 2.7: Request-response-interaction using `Message` objects.

We have implemented additional `RequestMessage` and `ResponseMessage` classes extending the `Message` class. A request message contains user information, while a response message features additional information about the processing of the request. We present these messages in Sect. 3.3.

2.5 Protocol Schema Definitions

The fifth component of our interaction architecture deals with the string representation of `Message` objects. It defines schemas — according to which `Data` and `Message` objects are encoded with XML elements — and classes that allow conversions of any type of object from and to XML elements.
Throughout this work we use the Jdom XML library [7] to implement the XML functionality. When we use the term *element* we refer to the class `org.jdom.Element`.
Figure 2.8 is a graphical schema definition for XML documents created by `Message` objects. The root element `message` contains one element called `data` which is generated by the member `Data` object to represent itself. The XML representation of the `Data` object conforms to a separate schema definition. In Fig. 2.9 we give an example XML document conforming to the schema definition for message representation. The content of the `Data` object element is collapsed. Such a document is sent as string value between local and remote handlers.

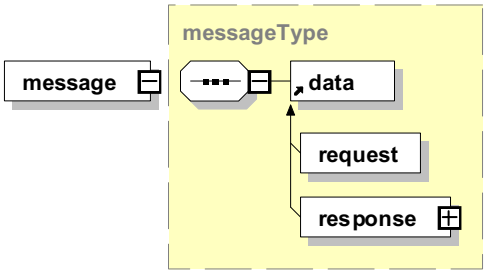

Figure 2.8: XML schema definition for the XML representation of a `Message` object.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <data class="org.ximtec.iserver.p2p.architecture.data.Response">
</message>
```

Figure 2.9: Example XML document representing a `Message` object according to the schema definition.

A `Data` object creates the XML element representing itself. For this purpose it follows a schema in which elements representing its member objects are assembled into one root element. The root element representing a `Data` object contains an attribute holding the class name of the data object. This is a requirement from the `Message` class that uses Java Reflection to reconstruct its member data. The `Data` class also has a `toNonJavaXML(String)` method which takes the result of `toXML(String)` containing the class name and removes this attribute. A `Data` object cannot be reconstructed from such an element but if this is not a requirement we can avoid including unnecessary information.
The `Request` class is an abstract class and thus its schema is only defined for its implementations. We will present a request schema definition with the implementations of our

architecture for iServer in Chapt. 3.
The response schema is shown in Fig. 2.10. A root element contains a child element representing the member object. The child element for the member is generated by the marshaler class for the member object type. Figure 2.11 is an example element representing a response containing an integer object.

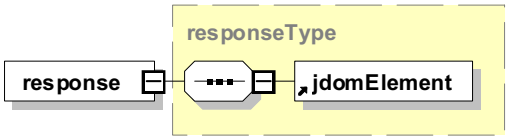

Figure 2.10: XML schema definition for a response element.

```
<?xml version="1.0" encoding="UTF-8" ?>
<response class="org.ximtec.iserver.p2p.architecture.data.Response">
  <integer>3</integer>
</response>
```

Figure 2.11: Example XML document representing a response according to the schema definition.

Common to all data implementations is the fact that they must all be able to obtain XML elements representing their member objects and to create the member objects from XML elements. These conversions from an object of any type to an XML element and vice versa are carried out by an XML element factory. The factory offers the two methods `fromXML(Element)` and `toXML(Object)` to support these conversions.
For each class of object that is to be represented as XML element and to be reconstructible from an XML element, there must exist a class that extends `org.jdom.Element`. This class offers a constructor that takes the object to be represented as XML element. It also has a static method taking an XML element as argument and returning a corresponding object. This method is called `unmarshal(Element)`. We refer to this class extending the `Element` class as marshaler for a particular class of objects. In Fig. 2.12 we give a pseudo UML definition for marshaler classes `Jdom[ Class ]Element` where the square brackets denote any object type name. For example, if we want to have integer objects as members of `Data` objects, we implement a `JdomIntegerElement` class that has a constructor taking an integer object as argument and a static method taking an XML element and returning an integer object.
We have implemented marshaler classes for Java primitive types and `String` objects. Figure 2.13 shows the XML schema definition for these marshalers. Their content is of type `String` and they do not have any attributes. More marshaler classes specific to iServer are presented in Sect. 3.4. Figure 2.14 shows an example XML document as generated by a `JdomIntegerElement` object.
The XML element factory is basically no more than a registry (`Hashtable`) for object

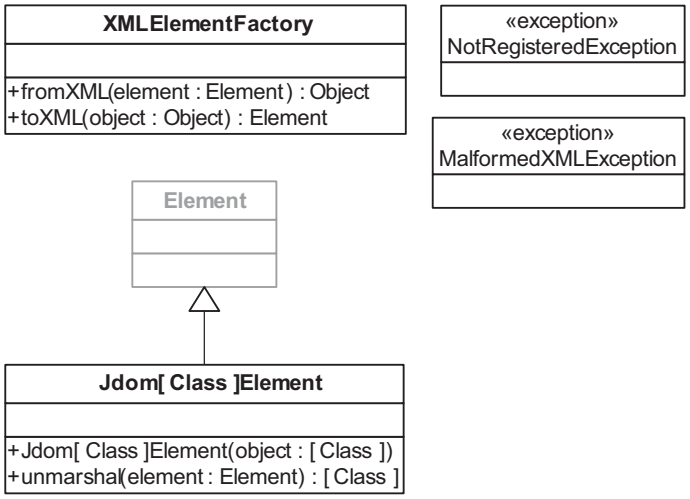

Figure 2.12: XML element factory and pseudo marshaler class.

classes and their marshaler classes. When the method `toXML(Object)` is called, the factory looks up the marshaler class registered for the type of the object given as argument and returns a new instance of it representing the object as XML element. The method `fromXML(Element)` looks up the marshaler registered with the name of the XML element and returns the result of the static `unmarshal(Element)` method invoked on this class. The look up key for the retrieval of the marshaler class when marshaling is the class name of the object to be marshaled. For unmarshaling an XML string into an object, the marshaler class is retrieved with the XML element name. Since the element name does not necessarily correspond to the class name we need to register marshalers with both keys. The factory throws a `NotRegisteredException` if a marshaler class cannot be found for a given key.
Figure 2.15 illustrates the process of marshaling and unmarshaling initiated by `Data` objects requested to convert themselves to and from XML elements by the message object. For each member object that is to be added to the root element representing the `Data` object, the factory is used to get the representing XML element (b). Similarly for unmarshaling, the `Data` object uses the factory to unmarshal each element representing an object (a) and assigns it to the respective member.

2.6 Assembly of the Components and Evaluation of the Architecture

Figure 2.16 summarises our interaction architecture graphically. The time line goes from top to the bottom. In a local system the request factory is used to create a `Request` object which is handed over to the local request handler. The handler creates a `Message` object containing the request and sends its XML string representation to the remote request handler. There the message is reconstructed from the string value received and the `Request` object extracted. The remote request handler initiates the processing of the request which is carried out by the

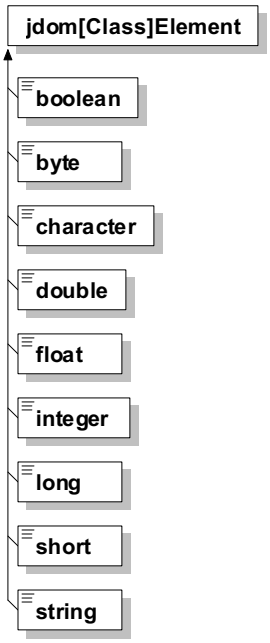

Figure 2.13: Marshaler classes for Java primitive types.

```
<?xml version="1.0" encoding="UTF-8" ?>
<integer>3</integer>
```

Figure 2.14: XML representation of a `JdomIntegerElement` object.

`Request` object itself. The resulting `Response` object is wrapped in another `Message` object and its XML string representation is returned to the local response handler. Note that in this constellation we use a response handler handling the response instead of letting the local request handler receive and return the response message itself.

2.6.1 Essentials of the Components

In this section we outline the advantages of using our interaction architecture. We point out the essential details of each of the five components which make a service implemented according to our architecture flexible and extensible.

Request Factory

The request factory must create objects that are subtypes of the `Request` class. The `Message` class used to wrap the request and the remote request handler initiating the re-

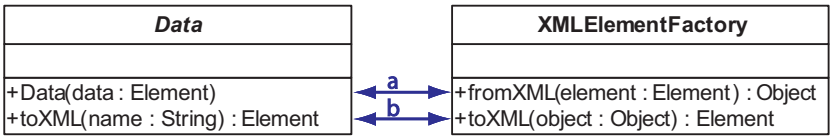

Figure 2.15: Interaction between `Data` objects and the XML element factory.

quest processing rely on the methods specified by the `Data` and `Request` classes. As long as the requests created by the factory comply to these specifications, all other components are not affected by any changes to the factory.

Handling

Handlers rely on the methods specified by the `Request` and `Message` classes only. This makes them independent both from possible subtypes implementing arbitrary requests and from the database API. Handlers take care of service framework specific requirements such as locating a remote handler and transmitting XML strings. Thus, we can create any service by implementing respective handlers that define interface methods using `Data` and `String` parameter and return types. If this condition is fulfilled, all other components are independent from the service used for interaction.
Handlers are also responsible for collecting the `Data` objects required to construct a `Message` object and to extract and handle this information before initiating the request processing. They must therefore be adapted if specialised messages containing additional information are used.
For example, we might want to transmit user information along with a request. Since a local response handler is most likely to be used by one user only, we could implement its constructor taking the user information as argument and assigning it to a private member. The user information would not have to be provided with every request and the handler could automatically transmit it along with any request. The handler would use a specific `Request-Message` class that has additional functionality to add and extract the user information. The remote request handler would be adapted to use the same specific message class to reconstruct a `Message` object from the XML string received and to handle the additional user information.

Request and Response

The `Request` and `Response` classes encapsulate the interaction with the remote information system. Implementations specific to a particular database API and request pattern must implement the methods specified by the `Request`, `Response` and their superclass. If this condition is satisfied, any request and response implementation can be used, exchanged and extended without any further adaptations of other components. We can transmit additional information to the remote system by implementing specific `Data` and corresponding `Message` classes.
In the previous example — where we supply the remote system with additional user infor-

mation when transmitting a request — we would create a new subtype `User` of the `Data` class since the user object is to be transmittable using the `Message` class. This new subtype contains all the user information as members and uses the XML element factory to convert them from and to XML elements when requested.

Messaging

The messaging component is responsible to convert a `Request` and `Response` object from and to an XML string. The handlers use this functionality to transmit `Data` objects. The handlers mainly rely on the `toXMLString()` method and `Message(String)` constructor to send and receive `Data` objects. A message contains all the information that is transmitted to a remote system. If we want to send additional information we implement subtypes of the `Message` class containing additional members of type `Data` and corresponding constructor and accessor methods.
To pursue our previous example of providing the remote system with additional user information we describe the respective `Message` subclass. We would implement a `RequestMessage` class extending the `Message` class and thus ensuring the availability of the `toXMLString()` method and `Message(String)` constructor. The `RequestMessage` class takes the user information as additional argument of its constructor and defines a corresponding member. The user information must be a subtype of the `Data` class to ensure to and from XML conversions.

Protocol Schema Definitions

The protocol schema definitions and XML element factory formalise the conversion from and to XML elements and string values. The factory and marshaler classes allow a `Data` class to define arbitrary types of objects as members and still be able to easily convert itself from and to an XML element. The XML schema according to which members of `Data` classes are converted can be exchanged by adapting the respective marshaler class. Such changes do not entail adaptations of any other components in our architecture.
The `Message` class creates its representing XML document and assembles the elements representing its member objects according to its XML schema definition. It relies on this schema for its reconstruction from an XML string. The creation of new subtypes of the `Message` class requires the designing of new XML schema definitions and changes to an XML schema definition require adaptations to the `Message` class only. All other components stay unaffected by changes to XML schema definitions.
To complete our example with the additional user information we name the adaptations required to the protocol schema component of our architecture. Firstly, we have a new subtype of the `Data` class containing the user information as member. Since this subtype must be able to generate an XML element from which it can reconstruct itself using the XML element factory, we would have to implement the marshaler class specific to its member types and register it with the factory. Secondly, we have a new subtype of the `Message` class which requires XML schema definitions. According to these schema definitions, the root element representing a `Request` message would now contain two elements, one representing the `Request` object and the other one representing the `User` object.

2.6.2 Possible Adaptations

To conclude this section we list all possible adaptations that can be conducted on a service implemented according to our interaction architecture:

Service

The service (e.g. web service, peer-to-peer service) can be changed by providing the handlers specifying interface methods that use `Data` and `String` parameter and return types.

Database API

The database API can be changed by providing subclasses of the `Request` and `Response` classes defined in our architecture. Additionally, the request factory must be adapted such that it creates these `Request` objects.

Request Pattern

The request pattern can be exchanged by providing the `Request` classes that implement the abstract method `computeResponse()` method. The request factory must be adapted so that it creates these `Request` objects.

Database Schema

The schema describing the information entities and relationships stored in an information system can be changed by adapting the request factory.

XML Schema Definitions

Changes to the XML representation of objects transmitted can be performed by adapting the marshaler classes. No further adaptations are required.

Exchange of Additional Information

The most complex adaptations have to be carried out when additional information has to be transmitted between local and remote request handlers. First we need to define a subtype of the `Message` class that contains the additional members and declares respective constructor and accessor methods. Second we define the subclass of the `Data` class that wrap the additional information so that it can be wrapped in a message. In third place we must design the XML schema definition according to which the new `Message` and `Data` type is converted from and to XML representation. Finally, the handlers must be adapted to provide the additional information when constructing the `Message` object and to extract and handle the additional information after reconstructing the message.

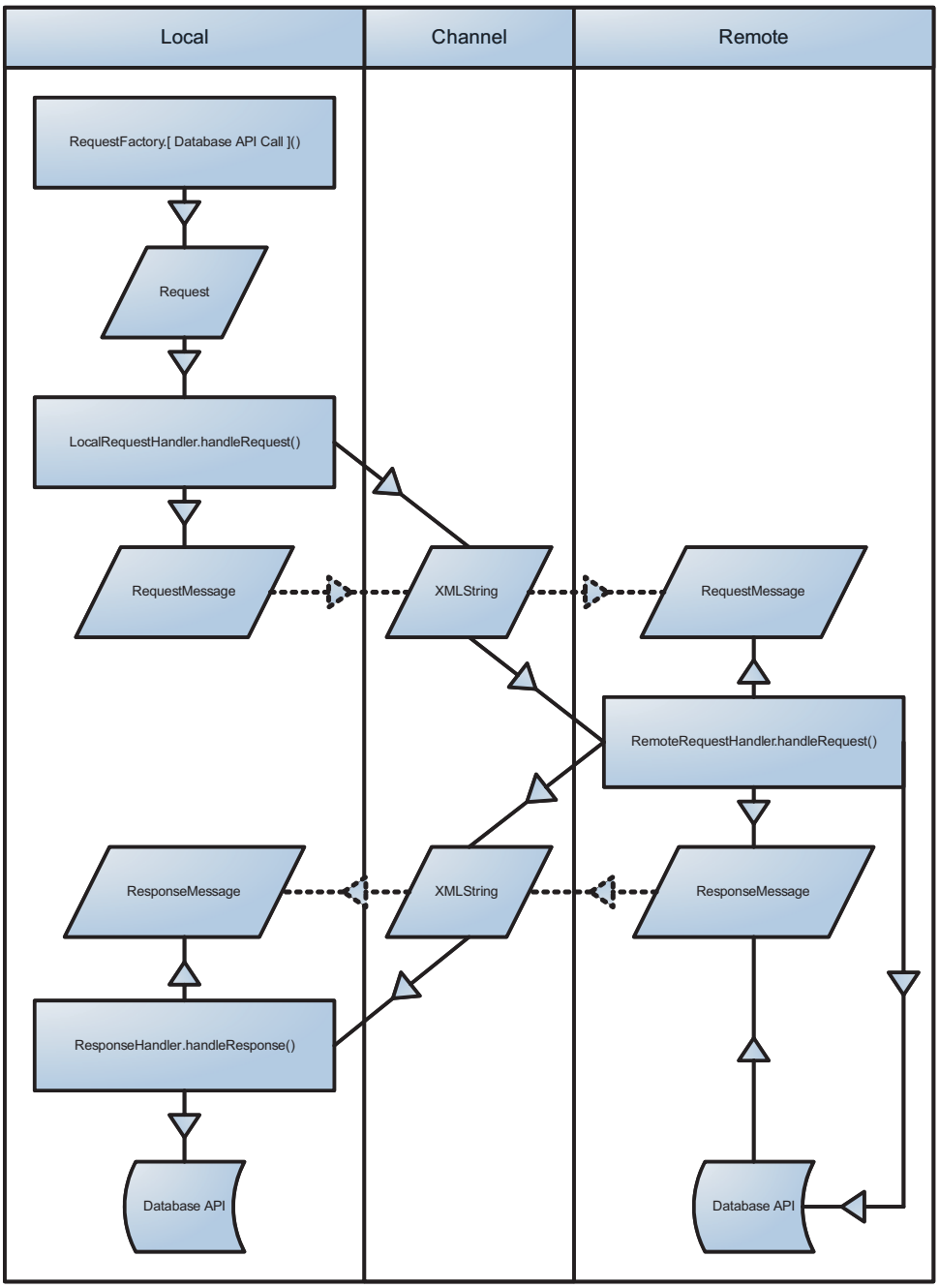

Figure 2.16: Workflow in a service implementing our interaction architecture.

3 Distributed iServer

In this section we introduce the implementation of our interaction architecture for iServer [14]. We present the details of each component. As we have stated in Sect. 2.2, handlers are specific to a particular service framework. In this section we describe a service framework independent implementation of a distributed iServer, thus, we present handler implementations for web and peer services in the respective Chapt. 4 and 5.

3.1 Request Factory

The request factory for iServer replicates the methods defined by the iServer API. For each of these methods there is a factory method returning a `Request` object representing the respective API method. The processing of a `Request` object returns a `Response` object containing the return value of the method invocation. Figure 3.1 shows the UML diagram of the `RequestFactory` class listing all supported iServer API methods.
A request factory can be implemented specific to arbitrary requirements offering a set of request methods. In this example factory every request corresponds to exactly one iServer API method. A request could also represent a higher level operation combining both arbitrary and multiple API method calls.
Parameter objects required to some of these factory methods will be sent to the remote request handler along with the request. Since we do not assume object identity in the local and remote iServer database, the parameter objects received by the remote request handler must first be matched with an object in the remote database without the use of object IDs. The matching procedure is encapsulated in the unmarshaler classes for iServer specific types such as `Entity`, `User` etc. The object created by the unmarshaler registered for its type is either an object contained in the remote database or a temporary object for which no matching objects could be found.

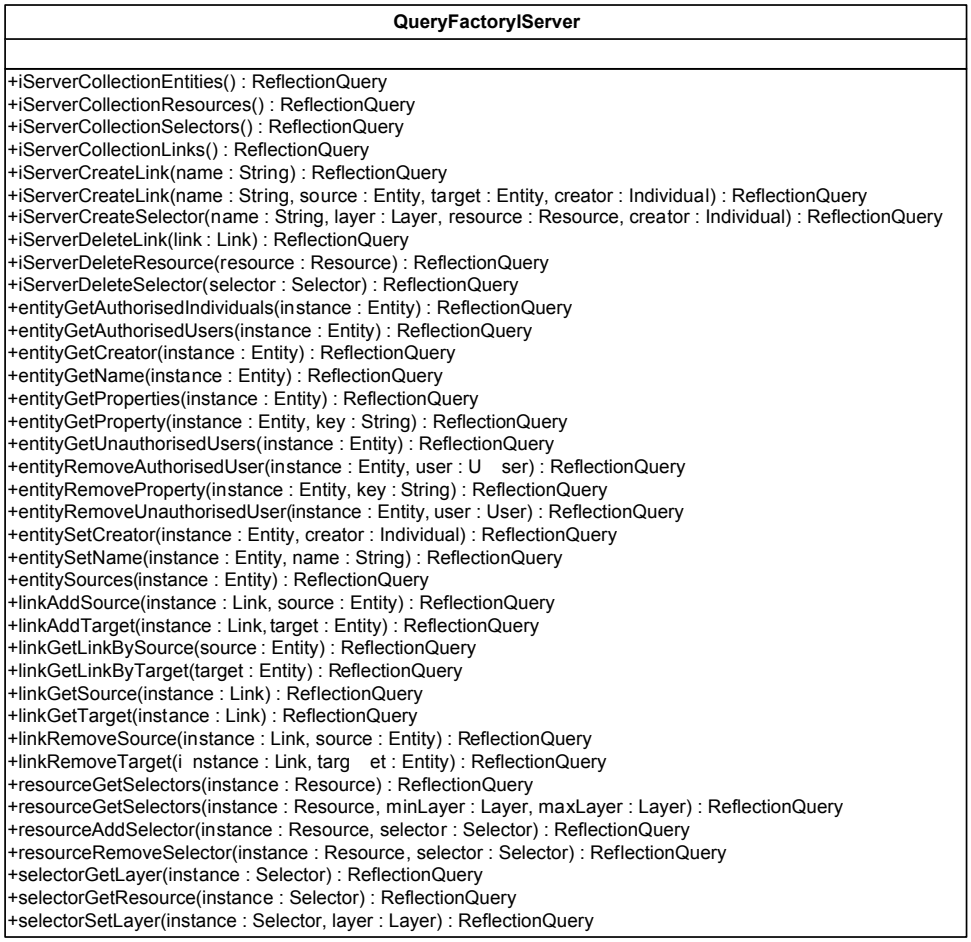

Figure 3.1: iServer Request Factory

3.2 Data

Figure 3.2 shows the UML diagrams of the abstract `Data` class and subtypes. The `Data` class specifies the methods required to generate an XML representation and to reconstruct a `Data` object from an XML element. Because it is impossible to define an abstract constructor method we defined an abstract `fromXML(Element)` method and implemented the `Data(Element)` constructor so that it invokes this abstract method.
Another implementation detail concerns the `Response` class. Since a `Response` object is designed to wrap an object of any type, the respective constructor takes an argument of type `Object`. Because the object to be wrapped may point to a null value, e.g. as a result of an iServer API method call, we have to consider the case when this constructor is called with a null value argument. In this case Java is unable to determine the constructor to be used by the argument object type and it chooses the most specific constructor which is the one for reconstruction from an XML element. To avoid this we define the constructor taking an object to be wrapped to have a second parameter of type boolean. In order to render this second parameter more useful than just for discerning the two constructor methods we use

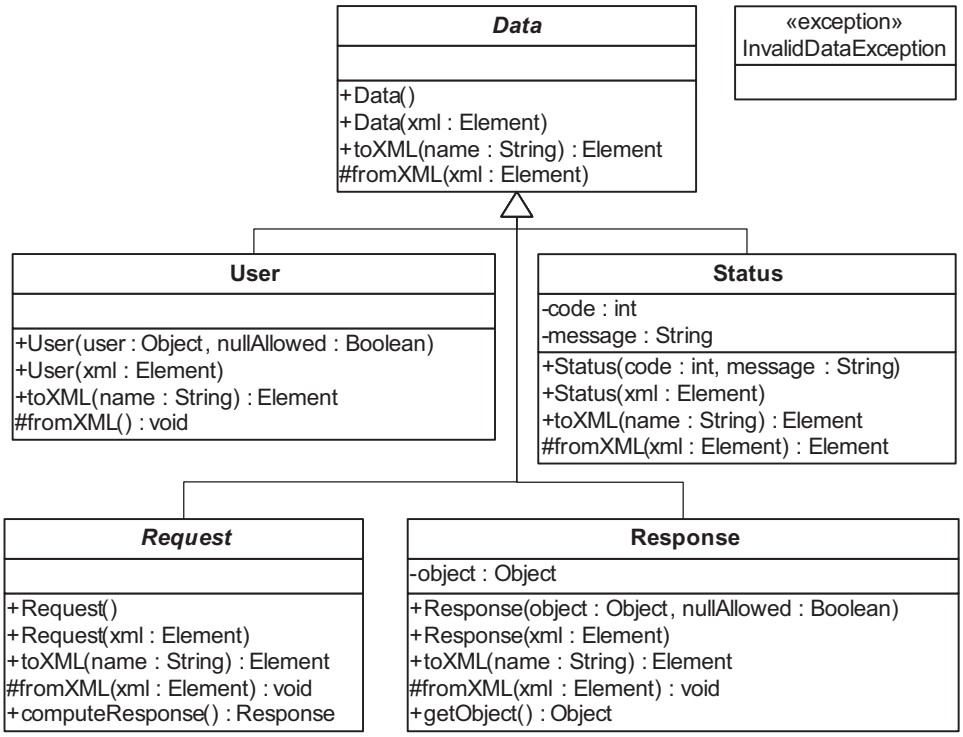

Figure 3.2: Data Implementation

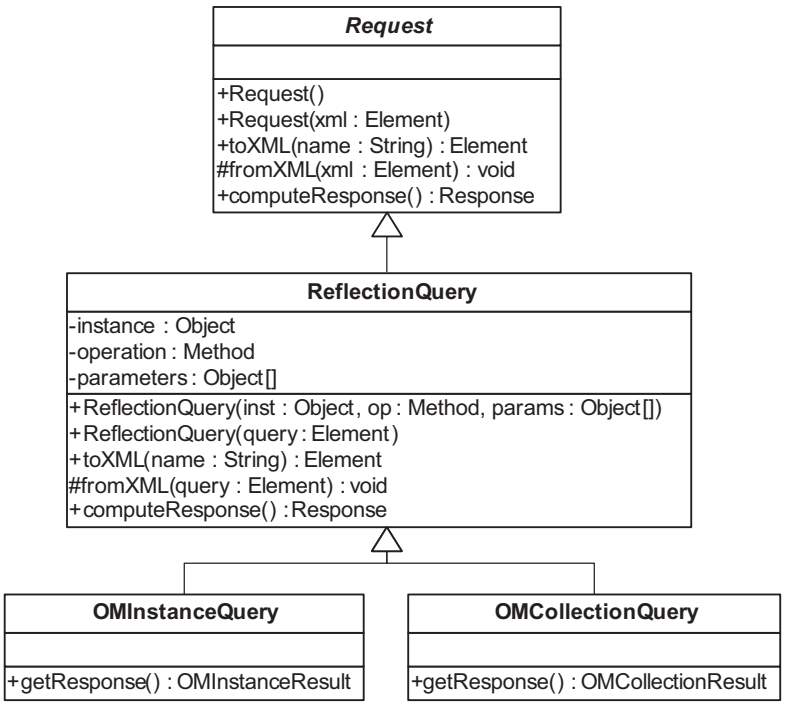

Figure 3.3: Request Implementation

the boolean value to determine whether a null value for the first parameter should be allowed. If the argument is true null values are allowed. If it is false the first argument is not allowed to be a null value and if it is, an exception is thrown. For all `Data` objects used to wrap the result of an iServer API method call this second argument is set to true.

In Fig. 3.3 we give the iServer specific subclasses of the abstract `Request` class. A `ReflectionQuery` object encapsulates all the data necessary to perform a method invocation using Java Reflection. This data consists of a method object, an array containing the parameter objects of the method and an object representing the instance on which the method is to be invoked. If the method is static the instance object points to a null value. If the method has no parameters, the array of parameter objects is of length zero.

The `ReflectionQuery` object can be constructed from a method object, instance object and an array of parameter objects. An XML element representing the query is generated by the `toXML(String)` method. A query can also be constructed from such an XML element. Finally, the method `processQuery()` executes the method invocation and returns its return value.

This class is a straightforward approach to represent an iServer API method call with a `Request` object. It can also be used to encode the invocation of an arbitrary method on any object. Thus, the request factory can construct a `Request` object representing any method call. We have also implemented another request pattern that is based on *selection* and *intersect* operations which can be nested. Since this approach circumvents the iServer API we no longer pursued it. Nevertheless, we discuss it in Appendix C.

In the following paragraph we explain the ReflectionQuery subtypes `OMInstanceQuery`

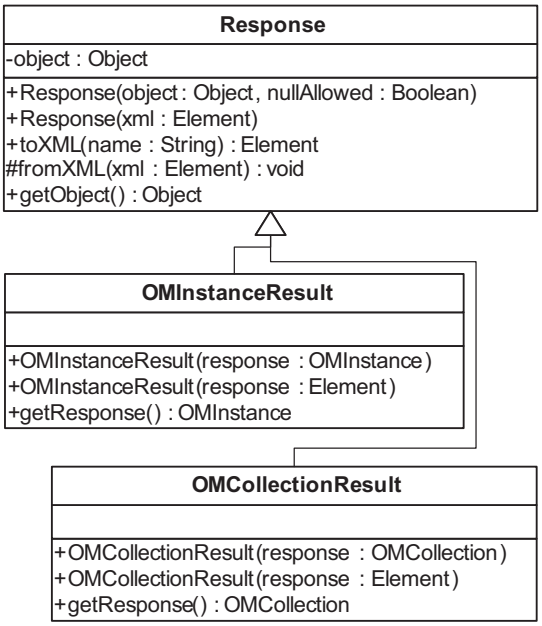

Figure 3.4: Response Classes

and `OMCollectionQuery`. For the response encoding we had to make only little extensions to the classes provided by our architecture. The `Response` class defines a member of type `Object` which is used for storing the `Response` object. iServer API methods return three different types of objects: String values, `OMInstance` and `OMCollection` objects. If the return value is of type `OMInstance` or `OMCollection` we want to avoid casting operations when accessing the `Response` object. Hence we implemented subclasses of the `Response` class that perform the cast inside the `getResponse()` method and return the object typed accordingly. The `OMCollectionResult` returns an `OMCollection` object and the `OMInstanceResult` returns an OMInstance object.
Since it is the `Request` object that constructs and returns the `Response` object, we created subtypes of the `ReflectionQuery` class specific to the return type of the method invocation. The `OMCollectionQuery` returns an `OMCollectionResult` object and the `OMInstanceQuery` returns an `OMInstanceResult` object. The request factory method constructing a `Request` object knows what type of object a particular request returns. Each factory method constructs a `Request` object corresponding to the result type of the iServer API method it represents.
In the next section we present request and response messages that extend the `Message` class. A request message provides the remote request handler with user information and the response message contains information about the outcome of the request processing. This information requires additional members to the one defined in the `Message` class. Since these members must be convertible to and from XML elements we implemented two additional classes extending the `Data` class. Figure 3.2 shows the UML diagram for the `User` and `Status` classes. At the moment a `User` object contains an object of any type e.g. a string value representing a name. For a full integration with the iServer user management

this object should point to an iServer user. For the same reason as in the `Response` class its constructor taking a parameter of type `Object` has the additional boolean argument. In the iServer peer service introduced in Chapt. 5 `User` objects are compared to avoid multiple responses by the same user. The `User` class thus overrides the `equals(Object)` method by delegating the equality request to its member object representing the user. A `Status` object contains an integer value representing a code, e.g. zero for successful processing etc. and an XML string representing a message.

3.3 Messaging

We implemented additional classes extending the `Message` class. The reason is that we wanted to provide the remote request handler with user information when sending a request. Also, the message containing the response should be enriched with status information about the outcome of the request processing. In the previous section we introduced the respective `Data` subclasses. In this section we present the implementation of the `Message` class as well as the `UserMessage`, `UserRequestMessage`, `ResponseMessage` and `User-ResponseMessage` classes. All classes are shown in Fig. 3.5.
For the reason given in Sect. 3.2, we added an abstract `fromXML(Element)` method to the `Message` class that allows to stipulate a constructor taking an XML element as argument within the `Message` superclass while implementing the construction from an XML element within its subclasses.
The `UserMessage` class extends the `Message` with one additional `Data` member. Its constructor and the getter method for this member require this object to be of type `User` and cast the return value to this type, respectively.
In order to determine the amount of access granted to a user requesting a remote database, a `User` object must always be included in a request message. Hence, we implemented a `UserRequestMessage` that extends the `UserMessage` class. It defines an additional `getRequest()` method that casts the `Data` object returned by the `Message` superclass accessor to a `Request` type.
We implemented a response message extending the base message with information about the outcome of the request processing. The additional member is of type `Status` and a constructor and accessor methods are defined accordingly. Similarly to the request message, an accessor method `getResponse()` avoids a cast when retrieving the `Response` object.
Sometimes information about the user responding must be included in the response message as well, e.g. in a peer service where multiple peers may respond to a single request. Thus we implemented a `UserResponseMessage` class that contains a `User` object as well as the constructor and accessor methods ensuring that the `Data` member of the `Message` superclass is of type `Response`. Since Java does not support multiple inheritance, we decided to make it a subtype of the `UserMessage` and to replicate the constructor and accessor methods from the `ResponseMessage` class.

3.4 Protocol Schema Definitions and Marshaling

Figure 3.6 shows the UML definitions for the XML element factory and the marshaler classes. In addition to the marshaler classes for primitive Java types presented in Sect. 2.5, we also im-

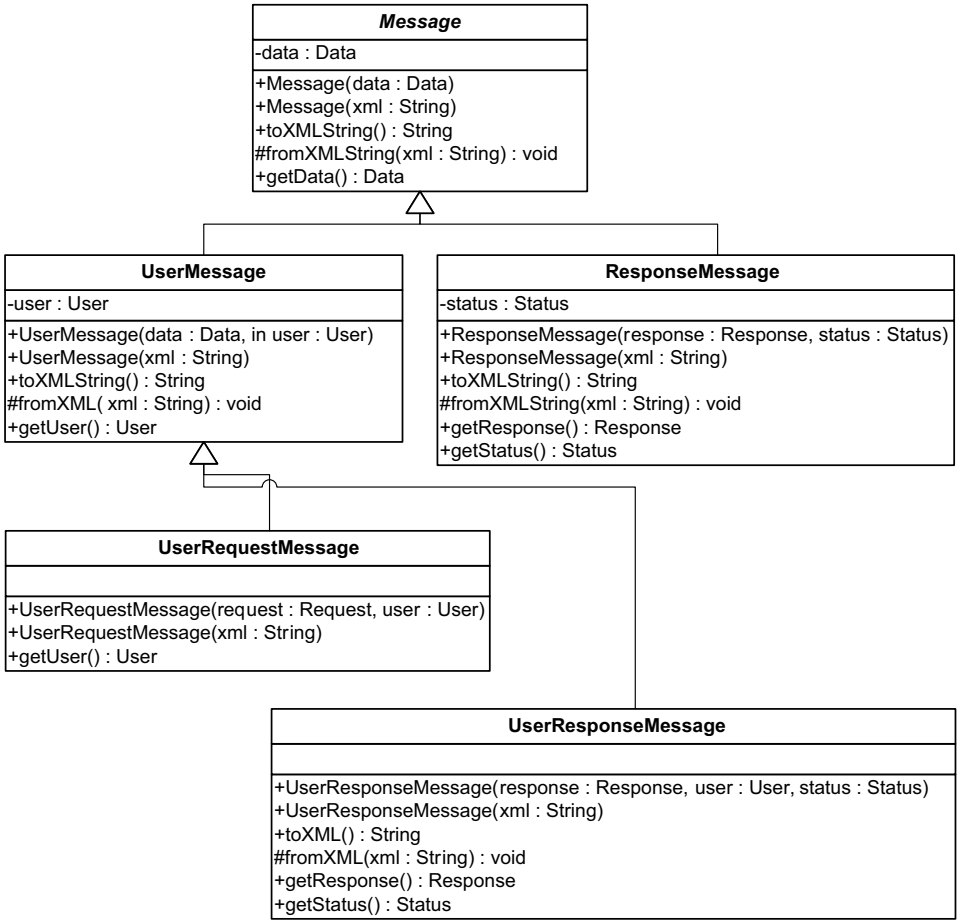

Figure 3.5: Messaging Implementation

plemented marshalers for `OMInstance` and `OMCollection` classes. Figure 3.7 illustrates the schema definitions for their XML representation.

The only additional functionality of our `OMInstance` marshaler compared to the ones defined within the iServer architecture (e.g. `JdomEntity`, `JdomLink` etc.) is that it returns an `OMInstance` object when unmarshaling as opposed to an `OMObject` object. Our marshaler uses the `JdomFactory` implemented within iServer to generate an element representing the instance object. It wraps this element and stores the instance type name as an attribute value in order to be able to reconstruct it given the `OMObject` object returned by the iServer factory when unmarshaling. Figure 3.8 contains an example `OMInstance` element as generated by the `JdomOMInstanceElement` marshaler.

An `OMCollection` marshaler represents an `OMCollection` object by marshaling all contained `OMInstance` objects using our `OMInstance` marshaler. These elements are added to the root element representing the collection object. The root element also stores the member type of the collection as an attribute value. With this attribute the collection can be reconstructed and populated with the unmarshaled instance member objects. Figure 3.9 is an example element representing a collection containing two instance object representations.

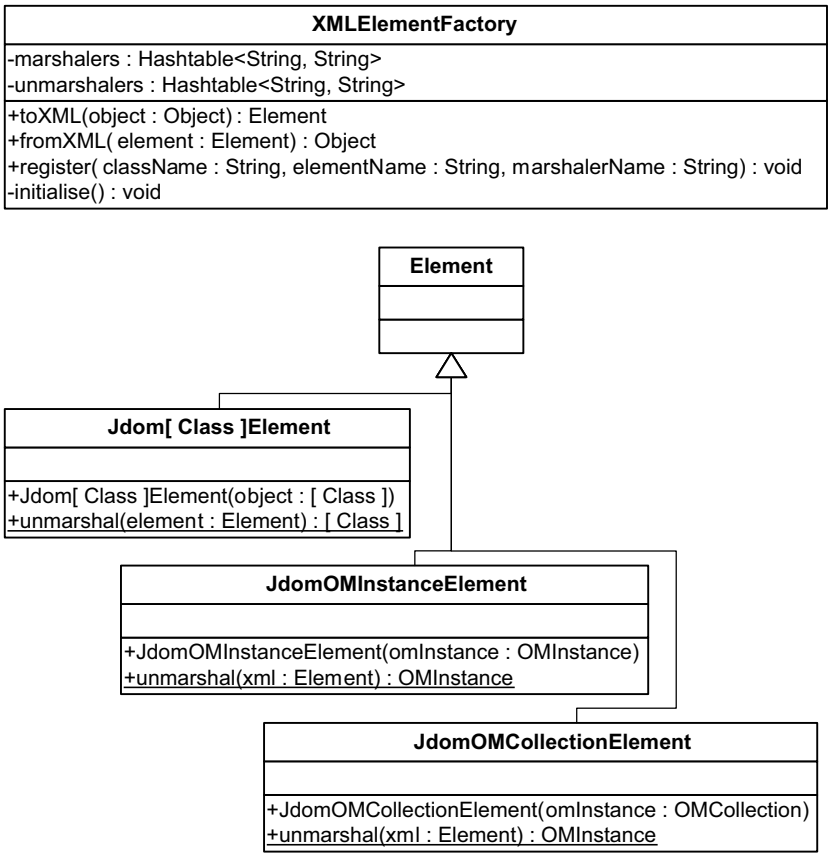

Figure 3.6: iServer specific implementations of the XMLElementFactory and Marshalers.

Figure 3.10 illustrates the XML schema definitions for XML representations of all subclasses of the `Message` class. Each subclass defines an additional member while it inherits the member(s) of its superclass(es). This inheritance hierarchy is reflected in the XML schema. In Fig. 3.11 we give an example XML document representing a user request message conforming to its schema definition. The `Data` object represented is a `ReflectionQuery` object. The user is represented with a string value containing his name. Both child elements are collapsed. Figures 3.12 and 3.13 are example XML representations of a response message and user response message, respectively. All child elements are collapsed.
Now, we define the schema for the XML representation of `ReflectionQuery` objects. Figure 3.14 graphically depicts the XML schema definition. The root element has three child elements: An `instanceObject` element contains the instance object on which the method is to be invoked as generated by our element factory. The names of the parameter classes are stored in a second element `parameterClasses`. A third element `parameterObjects` contains the parameter instances as generated by our element factory. The name of the class of the object on which the method is to be invoked and the method name are stored as attributes of the root element. Figure 3.15 shows an example XML representation of a reflection query object. The method to be invoked is called `intValue()` and it is declared by the `Integer` class. The method is to be invoked on an instance representing the integer value *3*.

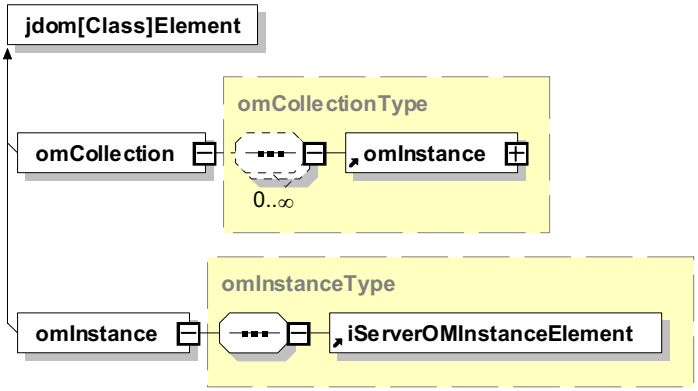

Figure 3.7: XML schema definition of marshaler classes for OMInstance and OMCollection objects.

At the moment, a `User` object is represented in XML as defined in the schema definition illustrated in Fig. 3.16. The root element contains one child that represents the member of type `Object` as generated by its marshaler class. Figure 3.17 shows an example XML document representing a `User` object. Its member object is of type `String` and contains the name of the user.
The XML representation of a `Status` object follows the schema definition shown in Fig. 3.18. The root element contains two child elements representing an integer and string, respectively. The `JdomIntegerElement` and `JdomStringElement` classes are used to marshal and unmarshal these members. Figure 3.19 is an example XML representation of a `Status` object. The integer is set to zero and the string contains a message.

```
<?xml version="1.0" encoding="UTF-8" ?>
<omInstance omInstanceName="entity">
  <entity>
    <name>African Savannah</name>
    <creator>
      <individual>
        <name>Beat Signer</name>
        <description />
        <login>signer</login>
        <password>signer</password>
      </individual>
    </creator>
    <authorised />
    <unauthorised />
    <properties />
  </entity>
</omInstance>
```

Figure 3.8: Example XML document representing an `OMInstance` object created by its marshaler class.

```
<?xml version="1.0" encoding="UTF-8" ?>
<omCollection omTypeName="entity">
  <omInstance omInstanceName="entity">
  <omInstance omInstanceName="entity">
</omCollection>
```

Figure 3.9: Example XML element representing an `OMCollection` object as generated by the `OMCollection` marshaler.

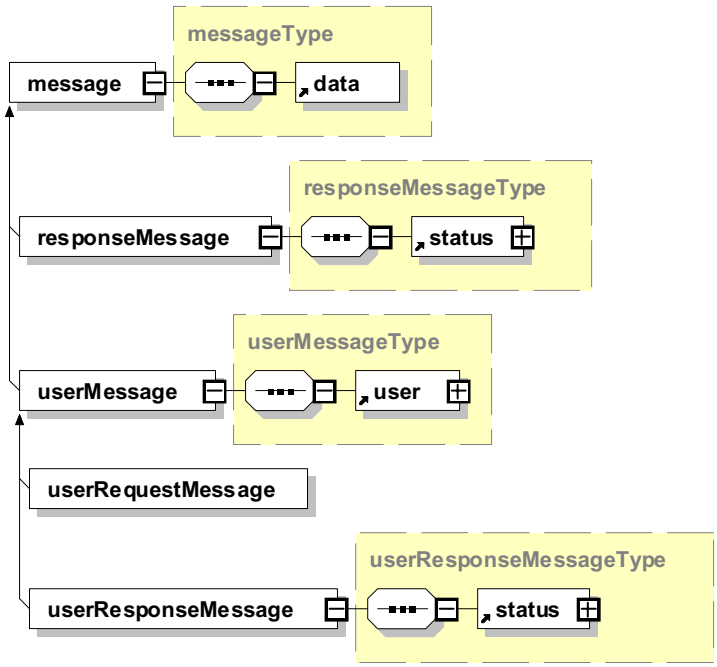

Figure 3.10: XML schema definitions for elements representing a user subtype implementations of `Message`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <user class="org.ximtec.iserver.p2p.data.User">
  <data class="org.ximtec.iserver.p2p.data.ReflectionQuery"
    className="java.lang.Integer" methodName="intValue">
</message>
```

Figure 3.11: Example XML document representing a user request message.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <status class="org.ximtec.iserver.p2p.data.Status">
  <data class="org.ximtec.iserver.p2p.architecture.data.Response">
</message>
```

Figure 3.12: Example XML document representing a response message.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <user class="org.ximtec.iserver.p2p.data.User">
  <status class="org.ximtec.iserver.p2p.data.Status">
  <data class="org.ximtec.iserver.p2p.architecture.data.Response">
</message>
```

Figure 3.13: Example XML document representing a user response message.

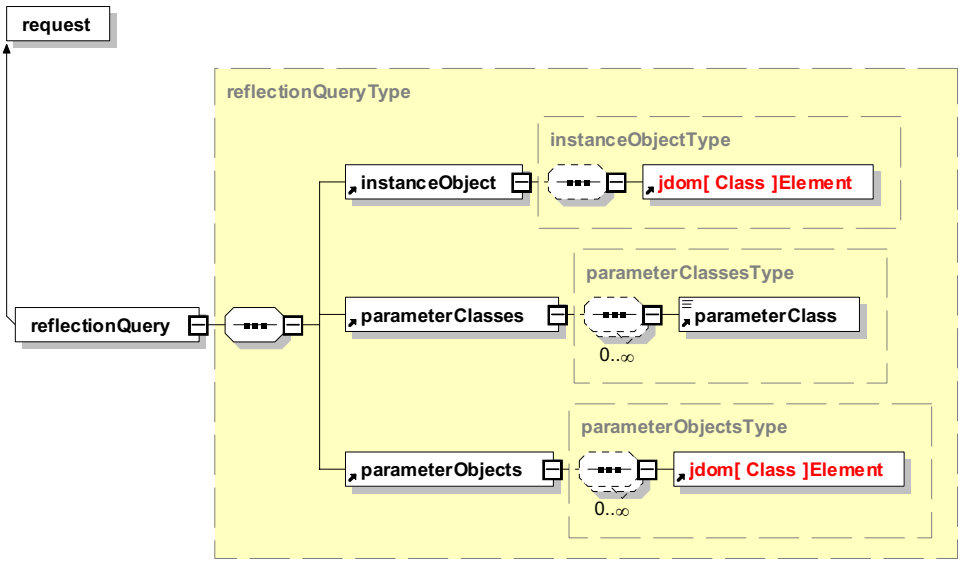

Figure 3.14: XML schema definition for elements representing a reflection query object.

```
<?xml version="1.0" encoding="UTF-8" ?>
<request class="org.ximtec.iserver.p2p.data.ReflectionQuery"
  className="java.lang.Integer" methodName="intValue">
  <parameterClasses />
  <parameterObjects />
  <instanceObject>
    <integer>3</integer>
  </instanceObject>
</request>
```

Figure 3.15: Example XML element representing a reflection query object.

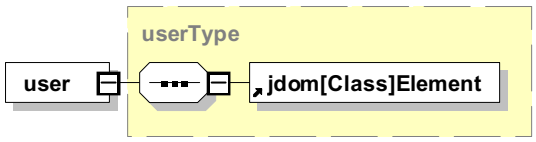

Figure 3.16: XML schema definition for elements representing a `User` object.

```
<?xml version="1.0" encoding="UTF-8" ?>
<user class="org.ximtec.iserver.p2p.data.User">
  <string>Vector Covar</string>
</user>
```

Figure 3.17: Example XML document representing a `User` object.

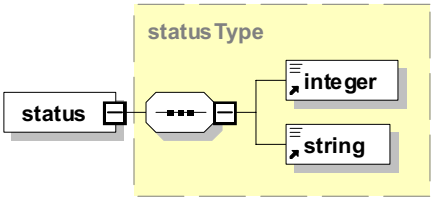

Figure 3.18: XML schema definition for elements representing a `Status` object.

```
<?xml version="1.0" encoding="UTF-8" ?>
<status class="org.ximtec.iserver.p2p.data.Status">
  <integer>0</integer>
  <string>request processed successfully</string>
</status>
```

Figure 3.19: Example XML document representing a `Status` object.

4 Implementation of an iServer Web Service

In this section we present our implementation of a web service for iServer. We outline the technologies we use and point out the components of our architecture that have been customised for web services.

4.1 Axis Web Service Framework

The axis web service framework can be regarded as a formalised employment of Java RMI technology [8]. The programmer avoids creating the necessary classes and interfaces such as the skeleton, stub, service locater etc. The axis framework is able to create the classes necessary for an implementation of a client server application based on RMI. The classes are created from the interface definition of a web service implementation. We include a step by step example implementation of a web service in Appendix B which is a good introduction to the Axis web service framework [1].
An axis web service runs as a servlet on a Tomcat server [2]. The service can be reached through a port set in the Tomcat preferences. A client uses the generated service locater class to create a local object allowing remote method invocations. Hence the web service is accessed with a regular method invocation in the client implementation.

4.2 Web Service Specific Components and Workflow

For a web service there is nothing to provide for the client side, hence, there is no web service implementation of a local request handler. A client has to implement the local handler functionalities on its own. The parameter objects of the iServer API method to be called must be prepared, marshaled and serialised. iServer API methods take either `OMInstance` objects or string values as arguments. For `OMInstance` arguments, the client must create an

XML string conforming to the XML schema definitions for a `JdomOMInstanceElement` object described in Sect. 3.4. String parameters are used without any further processing.
The result of a web service method invocation is a `ResponseMessage` object containing the `Response` and `Status` object. In Sect. 2.5 we describe the XML representation of `Data` objects which includes `Response` objects. In a web service we do not want to include any Java specific information into the XML representation of a response. Thus, the response is encoded without the attribute holding the class name of the `Response` object.
The only web service specific component is an implementation of the web service interface. There are two possibilities to make iServer API methods accessible remotely:

- The interface defines one general method taking an XML string representation of a `Request` object as argument. This parameter object uniformly encapsulates the method to be called which is why one interface method is enough to handle all API methods.
- The other possibility is to define an interface method for each API method available as web service. In this case there is no need to encode the method to be called since the client chooses the corresponding web service method directly. The second approach makes it easier for the client to use the web service because there is less XML code to be generated.

We decided to define and implement a web service method for each iServer API method offered as web service. The interface currently offers all methods defined in the iServer API. This is not a necessity and the service interface could offer access to the remote database consisting of any subset of the API methods.
Figure 4.1 shows the UML definition of the web service interface. There is a corresponding interface method for each iServer API method. This interface can be regarded as the web service specific implementation of a `RemoteRequestHandler` as defined in our interaction architecture. The difference is that the `handleRequest(Data)` takes `Data` objects as argument as opposed to the request handling web service methods. The similarity is that both return `Data` objects wrapped with a `Message` object.
Figure 4.2 shows the implementation of an example web service interface method. In a first step, the object on which the method is to be called is deserialised and unmarshaled from the XML string received as argument. If the method to be called requires parameter objects, these would be prepared in a similar way. Then the iServer API method is invoked and its return value wrapped with a `Response` object. The latter is marshaled and serialised using a `Message` object and its XML string representation is returned to the requesting client. Note that in this example we do not return a `ResponseMessage` object to keep the example code short.
Figure 4.3 schematically illustrates the interaction in a web service. The remote request handler — i.e. the web service interface — currently replicates the methods of the iServer API. The string values are serialisations of XML elements representing the parameter objects of an invoked web service method. An XML document is built from these XML strings in the handler method, i.e. in the implementation of the web service interface method that is invoked. The `XMLElementFactory` is used to unmarshal the element (a and b). The web service method retrieves the response using iServer API methods of the remote database (c and d). The result is wrapped with a `Response` object which is wrapped with a `Response-`

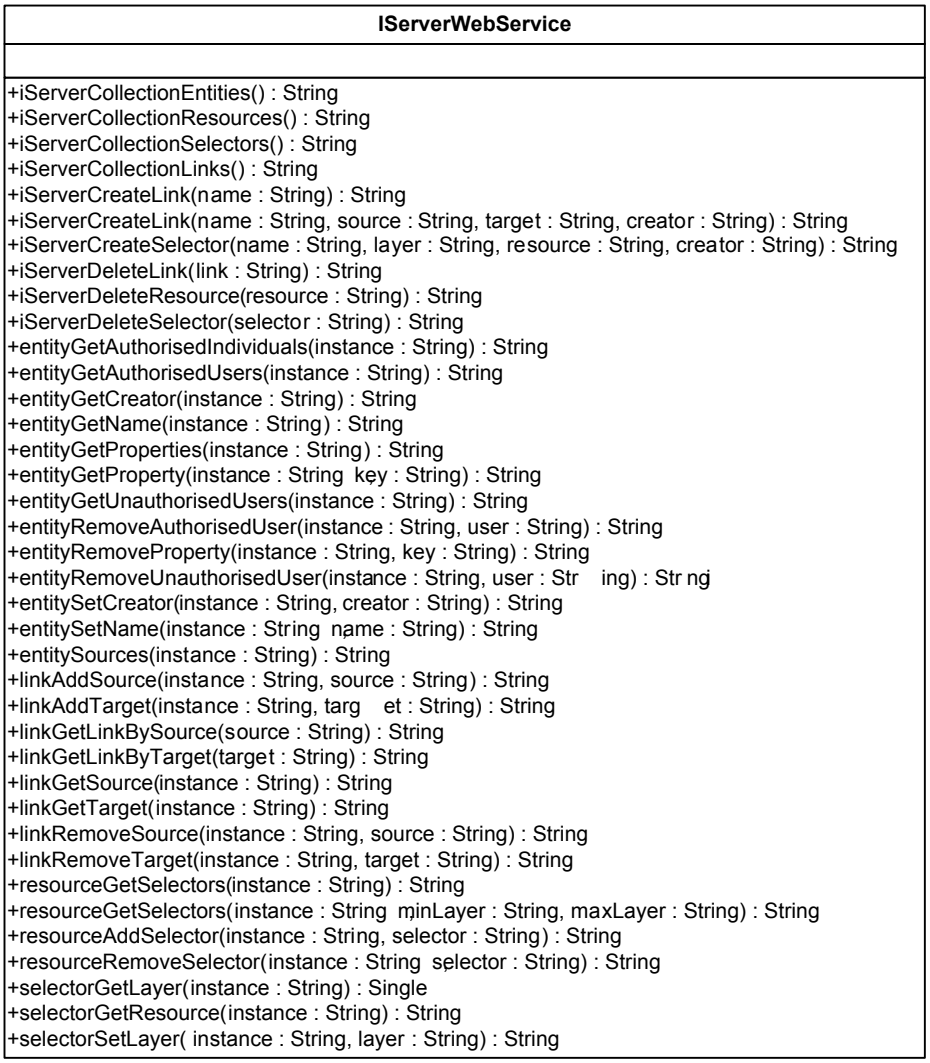

Figure 4.1: iServer Web Service Interface

`Message` object on his part. The XML string representation of this message is returned to the client (e).
Figure 4.4 illustrates the workflow initiated by a client calling a web service interface method. The time line goes from top to the bottom. The client provides the parameter objects represented as XML strings. The web service interface unmarshals the parameter objects using the XML element factory. Once all parameter objects are prepared, the respective iServer API method is invoked. The result is returned to the client as an XML string value.

4.3 Example Usage

Figure 3.8 shows a parameter XML string as provided by a client calling the web service methods representing the iServer API methods `Entity.sources()` and

```
public String entityGetCreator(String entityStr)        {

        // deserialise and unmarshal source
        Document doc = builder.build(entityStr);
        Element entityXML = doc.getRootElement();
        Entity entity =
                (Entity) XMLElementFactory.fromXML(root);

        // query DB
        Response res = entity.getCreator();

        // marshal, serialise and return response
        Message resMsg = new Message(res);
        return resMsg.toXMLString(false);
}
```

Figure 4.2: Example web service method implementation.

`Entity.getCreator()`. Both web service methods take one parameter of type `OMInstance`, i.e. an entity object.
Figure 4.5 shows the message returned for the invocation of the `entityGetCreator(String)` method. The response message contains the creator object wrapped in a response element. The response message also contains a `Status` object.
Figure 4.6 contains the response to the invocation of `entitySources(String)`. The returned message consists of a `Status` and `OMCollection` object containing two `OMInstance` objects of type `Entity`.
Figure 4.7 shows the graphical user interface of the TCP monitor class that ships with the axis library. The monitor can be set to listen on a particular port and to forward the intercepted messages to another port. Thus it can be put in between a web service and its client to capture and display all messages passed. Web services send SOAP messages containing the argument and return values of the web service method. Though our request and response messages are formated in XML they are treated as string values, hence all brackets and quotes are encoded using character entity references such as $\&quot$; and $\&lt$; etc.
Further information about how to get a web service started and running are given in Appendix A.

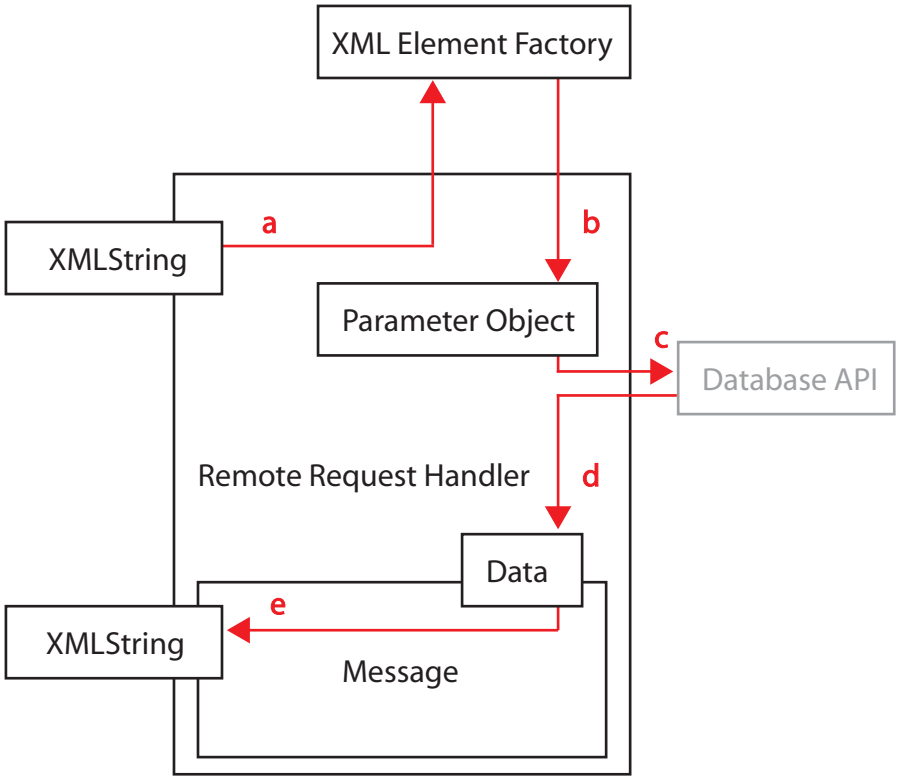

Figure 4.3: Schematic illustration of the interaction in a web service.

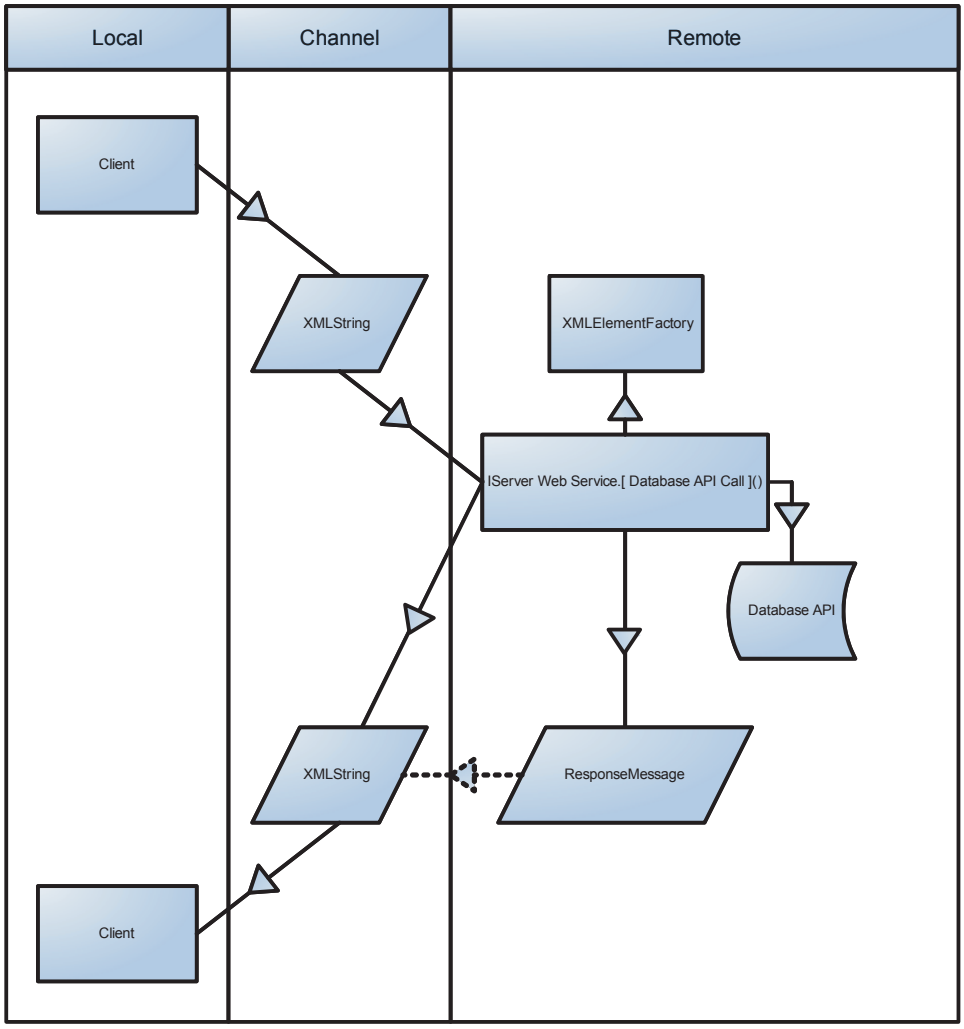

Figure 4.4: Workflow for iServer web service.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <status>
    <integer>0</integer>
    <string>request processed successfully</string>
  </status>
  <data>
    <omInstance omInstanceName="individual">
      <individual>
        <name>Beat Signer</name>
        <description />
        <login>signer</login>
        <password>signer</password>
      </individual>
    </omInstance>
  </data>
</message>
```

Figure 4.5: Response message returned by the web service method `entityGetCreator(String)`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <status>
    <integer>0</integer>
    <string>request processed successfully</string>
  </status>
  <data>
    <omCollection omTypeName="entity">
      <omInstance omInstanceName="entity">
        <entity>
          <name>Background Buffalo</name>
          <creator>
          <authorised />
          <unauthorised />
          <properties />
        </entity>
      </omInstance>
      <omInstance omInstanceName="entity">
        <entity>
          <name>Background Cheetah</name>
          <creator>
          <authorised />
          <unauthorised />
          <properties />
        </entity>
      </omInstance>
    </omCollection>
  </data>
</message>
```

Figure 4.6: Response message returned by the web service method `entitySources(String)`.

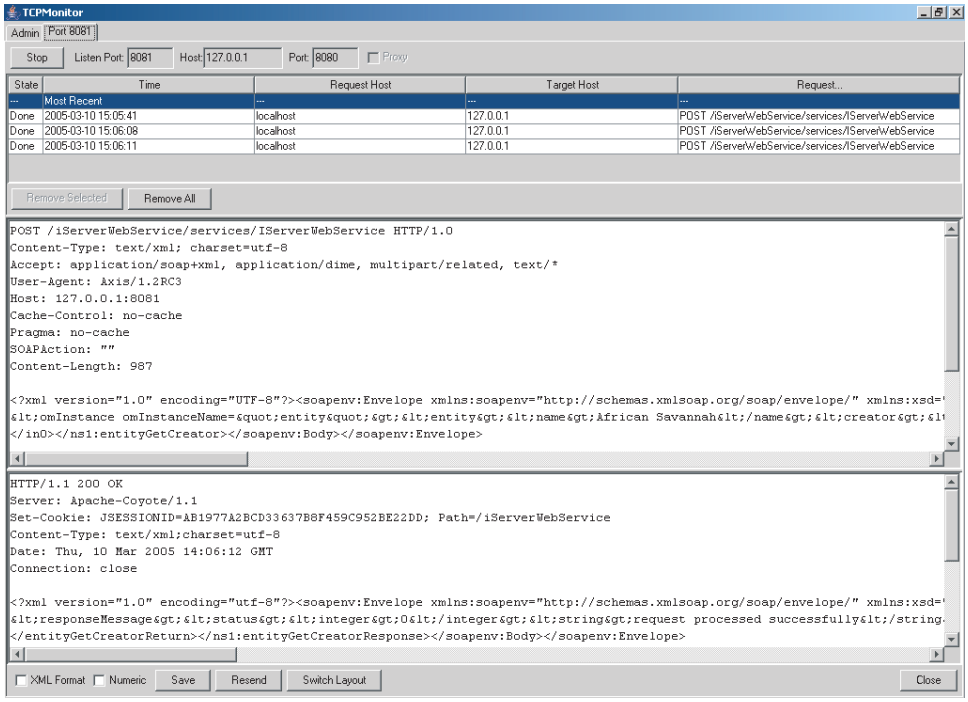

Figure 4.7: TCP Monitor GUI allowing to intercept the SOAP messages used within a web service.

5 Implementation of a Peer Service for iServer

The main goal of this thesis was to design and set up a peer-to-peer implementation of iServer. To do this we implemented peer-to-peer specific classes conforming to our architecture. In this section we present the technologies we used and the extensions/implementations made to our architecture. We also give usage examples by presenting code fragments relevant to the understanding of the functionality of our classes.

5.1 JXTA Framework

As proposed in the previous thesis on iServer peer-to-peer [6] we chose to implement our peer service using the JXTA framework [9]. Our knowledge about JXTA is based on the programmer's guide [11], Brendon Wilson's introductory book [16] and Java P2P Unleashed [5]. We give a quick introduction to the main concepts and terms in JXTA and refer to these sources for further needs.
JXTA is an open collaboration platform that supports most needs of a distributed and decentralised architecture. The achievement of the project is the definition of a set of language independent protocols that constitute a general purpose network programming framework. The protocols define the mechanisms for collaborating peers to communicate while any device connected to a network can participate by implementing one or more of them.
A JXTA network of collaborating devices is a set of interconnected nodes referred to as `Peers`: any device connected to the network and implementing one or more JXTA protocols is a peer. A community of peers that have agreed on a common set of services is called a `Peer Group`. The JXTA protocols define the manner in which peers perform, among others, the following tasks:

- Peer discovery

- Create, join and grant access to a peer group
- Advertise and discover services
- Communicate with other peers

JXTA peers advertise their services with XML documents called `Advertisements`. Peers discover services by retrieving advertisements that enable them to connect to and interact with the service providing peer. Peers use `Pipes` to send messages to each other. Pipes are encapsulated communication channels enabling point-to-point message exchanging. A peer may also broadcast messages to all members of a group while the reply to a broadcasted message is received by its originator only.
In the following sections we give some details relevant to our iServer peer-to-peer implementation about the main components of a peer service based on the JXTA framework.

5.1.1 Advertisement

An advertisement is a language-neutral metadata structure that describes peer resources such as peers, peer groups, pipes, and services. JXTA protocols use advertisements to describe and publish the existence of peer resources. Peers discover a resource by retrieving the respective advertisement. Among others, the JXTA protocols define the following advertisements: `Peer Advertisement`, `Peer Group Advertisement`, `Pipe Advertisement` etc. Advertisements are represented in XML documents.

5.1.2 Peer

A peer is a device connected to the network implementing at least one of the JXTA protocols. Peers send and receive messages and process them in order to establish a service. `Rendezvous` and `Relay` peers are special peers that implement particular protocols.
A rendezvous peer is responsible for propagating messages within a peer group. It maintains a cache of advertisements and forwards discovery messages to help other peers discover resources. When a peer joins a peer group, it automatically connects to a rendezvous peer. If none can be found, it becomes a rendezvous peer for this group. A rendezvous peer also manages a list of other known rendezvous peers.
A relay peer maintains information on routes to other peers and routes messages to peers. A peer first checks its own cache for route information to another peer. If nothing can be found, queries for route information are sent to relay peers. Relay peers also forward messages between peers that cannot directly address each other, e.g. because of firewall restrictions.

5.1.3 Peer Group

A peer group is a community of peers that agree on a common set of services. A group can be created and/or joined by peers. A group advertisement can be published to make it possible for peers to discover it. Peer groups define their own membership policy, e.g. *anybody can join* or *highly secured and protected*.
The `Net Peer Group` is a predefined group which all peers belong to by default. Groups form a hierarchical parent-child relationship in which each group has a single parent. The advertisement for a group is published in itself and its parent group.

Peer groups provide, among other, the following services to its members: `Discovery Service`, `Membership Service`, `Pipe Service`, `Resolver Service`.

5.1.4 Resolver Service and Resolver Query Handler

A resolver service is a service based on the `Peer Resolver Protocol`. This protocol enables peers to broadcast generic requests to all members of a peer group and to receive responses from replying peers. All peers receive the request but only the originator receives the replies. A resolver request can also be sent to a particular peer using its unique identifier. The resolver service is provided by the peer group to its members. Any peer can deploy a service by implementing the `QueryHandler` interface and registering it with the resolver service of the group. The query handler will be notified about requests by the resolver service. A service is identified by a unique identifier which is set upon registration and which is used to send a request to it. `Resolver Query Messages` and `Resolver Response Messages` are used to send requests and responses, respectively.

5.1.5 Pipe

A JXTA pipe is an encapsulated mechanism providing two peers with a channel through which they can send messages. Pipes are unidirectional and their endpoints are referred to as `Input Pipe` and `Output Pipe`: an `Input Pipe` is advertised by a peer listening for incoming messages and an `Output Pipe` is used by a peer sending a message. Pipe communication is asynchronous. The input pipe notifies its listeners waiting to receive a message about incoming messages while the output pipe notifies its listeners waiting to send a message about an already established connection.
Typically, pipes are used for point-to-point communication, but `Propagate Pipes` exist that connect one output pipe to multiple input pipes. `Secure Unicast Pipes` are point-to-point pipes that provide a secure communication channel.

5.2 User Management

In our iServer peer service a request is always broadcasted to all members of the group. Thus, a requesting peer receives multiple responses for a single request. We want to rate a response by the user sending it in order to filter the potential vast number of responses. Therefore, we implemented a user management that allows user rating. Additionally, we implemented a user validity system that maintains a list of valid users shared by all peers.

5.2.1 User Validity

Our peer group maintains a set of user validity tuples which every member stores locally. Such a tuple contains two elements of type `User` and `Boolean`. A user is valid if it is paired with true and invalid otherwise. A new user can be added to the set as a valid user if it is not already contained and set invalid. A user can be set invalid but never revalidated again. Note that, as opposed to the set of online peers, this set also contains users that may be not online.
All members of the peer group store the set of user validities locally. We give the steps necessary to keep these sets consistent:

- On startup a peer tries to read a file containing tuples stored in the previous session. If this file does not exist, it creates a new one.
- It then creates a local tuple set S_{local} containing the tuples in the file. If the user owning the peer it not contained, it is added to this set.
- Whenever a peer joins the group, it retrieves the set of user validities of all other members.
- Every incoming set S_remote is treated as follows:
  - If S_{local} contains all tuples in S_{remote} and every two corresponding tuples from each set either have the same boolean value or they differ and the local tuple contains a false value then leave the local set as it is. If S_{remote} contains a tuple not contained in S_{local} then add this tuple to the local set. If two corresponding tuples from each set have a different boolean value, and the local tuple has a true value then set it to false.
  - If S_{remote} contains all tuples in S_{local} and every two corresponding tuples from each set have the same boolean value or they differ and the remote tuple has a true value, then nothing more is done. Note that, so far, in no two corresponding tuples with different boolean values the boolean value of the local tuple can be true. In all other cases broadcast the local set of tuples to all other members of the group which each treat an incoming set as just described.
- When no more sets are broadcasted all tuple sets are consistent.
- As soon as any local set changes it is broadcasted to all other members of the group.

We do not use this set of user validities within the iServer peer service yet. We have implemented user validity management in conjunction with user rating where it is not used anymore. However, the set is kept consistent and it is left to future uses of iServer peer service to decide whether to keep maintaining user validity management or not.
Figure 5.1 contains the UML diagram for the `UserValidityVector` class implementing the functionality described above. We display the public methods only, i.e. one for adding and one for removing a user as well as a method to check whether a user is valid or not. Note that the `rmUser(User)` method just sets a user to invalid since removing a user is prohibited. A `UserValidityVector` object is created by providing it with a `UserValidityRequestHandler` object described in Sect. 5.4.2 which is used to automatically propagate changes to the local set of user validities.
The set of validities is saved to a local file by every peer so it persists even when no peer is running. When a `UserValidityVector` object is created it tries to read the set of validities from this file. In case that this file does not exist or cannot be read for any other reason, a new file is created. Any change to the set are stored immediately.

5.2.2 User Rating

A user rating is a tuple with four entries $(user_i, user_j, r_{ij}, t)$: the rating user i, the rated user j, the rating value r_{ij} and a timestamp t. Such a tuple is always created by the rating user and

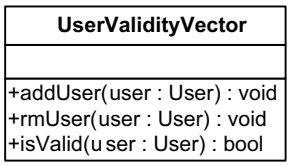

Figure 5.1: UML diagram for `UserValidityVector` class.

propagated to all other members of the group. Our user management ensures that every peer has the same set of tuples stored locally. This tuple synchrony is achieved as follows:

- On startup a peer tries to read a file containing tuples stored in the previous session. If this file does not exist it creates a new one.
- It then creates a tuple set S_{local} containing all tuples in the file. Whenever S_{local} changes a current snapshot is stored to the file.
- When the peer has joined the group it requests the tuple set S^i_{remote} from all other members i of the group.
- Every incoming tuple set S^i_{remote} from member i is compared with local set S_{local}:
  - If S_{local} contains all tuples in S^i_{remote} and all tuples in local set have a timestamp greater than or equal to the ones in the remote set, then leave the local set as it is. In any other case update the local set.
  - If S^i_{remote} contains all tuples in S_{local} and all tuples in remote set have a timestamp equal to the ones in the local set, then nothing is done. Note that no remote timestamp can be greater than in local set now. In any other case the local set is broadcasted to all other members of the group who proceed the same as described above.
- When no more sets are broadcasted all tuple sets contain the same tuples with equal timestamps each.
- Whenever a new rating is set locally the local set of tuples is broadcasted to all other members of the group each of which treat an incoming set as described above.

Whenever a user a receives a response to an iServer request from another user b this response is rated using the tuple set. If a tuple $(user_a, user_b, r_{ab}, t)$ exists then r_{ab} is the rating value. If such a tuple does not exist, we use sequences of users $(user_a, ..., user_i, user_j, ..., user_b)$ where there exist a tuple $user_i, user_j, r_{ij}, t$ for every two neighbors $user_i, user_j$ appearing in this sequence.
The idea can be described as follows: if a user a has not explicitly rated user b, but it has rated a third user i which has rated user b, then the tuples $(user_a, user_i, r_{ai}, t)$ and $(user_i, user_b, rib, t)$ can be aggregated to obtain an appropriate estimation of the rating value r_{ab}.

The tuple set can be regarded as a weighted directed graph (V, E) where a vertex $v_i \in V$ represents the user i and a directed edge $(i, j) \in E$ has the weight r_{ij} representing the rating value user i gives to user j. We treat the rating value as an amount of trust that flows from the rating user to the rated user. Thus, we can formulate the question of how a user i rates another user j as a well known graph theory problem: what is the maximum flow from a vertex v_i to another vertex v_j. For every path p from v_i to v_j we take the smallest weight and name it r_{ij}^p. All these weights r_{ij}^p are added up and returned as the amount of trust that flows from user i to user j.
The `UserRatingManager` object is created by providing it with a `UserRating-RequestHandler` object described in Sect. 5.4.2. This handler is used to automatically propagate new ratings. The class defines methods to set and get a user rating tuple. The timestamp is determined and added within the setting procedure. It uses a `Graph` [10] object to manage the rating values between pairs of users. The method `getUserRating(User, User)` first checks if there is a direct edge pointing from the vertex v_{rater} representing the first argument to v_{ratee}, the one representing the second. If this exists, the method returns its associated weight which represents the rating value. If there is no direct edge, an Edmonds-Karp [4] algorithm is run that returns the max flow value from v_{rater} to v_{ratee}. Note that if the graph is not connected and the two vertices do not belong to the same subgraph, then the returned max flow value is 0.
Figure 5.2 shows the UML diagram for the `UserRatingManager` class implementing the user rating management. We display the public methods only, i.e. one for setting/updating a user rating and the other one for retrieval of a user rating.
The set of user ratings is stored to a local file by every peer. Hence it also persists when no peer is running. When a `UserRatingManager` object is created it tries to read in the tuple set from this file. If this fails it creates a new one. Any changes to the tuple set are stored immediately.

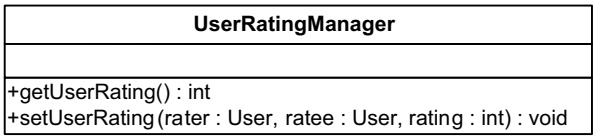

Figure 5.2: UML diagram for `UserRatingManager` class.

5.2.3 Propagation Retardation

Whenever a peer retrieves user validity/rating sets from all other members of the group it will receive several sets within a short period of time. Each incoming set possibly causes the propagation of the local set to all other members. This would produce a vast number of exchanged messages since all other peers potentially propagate as well.
To avoid this we have decided to locally delay the request for propagation for a fixed amount of time and to ignore all subsequent requests within this period of time. Since with every incoming set the local set becomes more consistent with the one containing all tuples, i.e. deletions are not supported, there is no problem in waiting for the last incoming set before prop-

agating the local set.
We implemented a class `ExecutionDelayer` as a thread taking the arguments necessary to make a method invocation using Java Reflection and a delay when constructed. The thread waits for the given period of time and then executes the given method. It also maintains a static set of `Method` objects currently pending for executions and no new instance of an `ExecutionDelayer` object delaying a method call already pending is created. This is achieved by setting the visibility of the constructor to private and defining a static `addDelayer(Method, Object, Object[], int)` method that creates a new thread only if the method is not contained in the set of pending methods. Figure 5.3 illustrates the UML definition of the class.

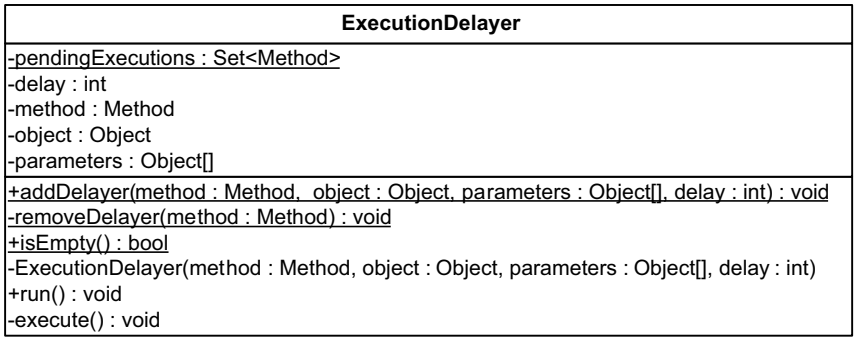

Figure 5.3: UML diagram for `ExecutionDelayer` class.

The `propagate()` method of the handlers described in Sect. 5.4.2 automatically delays its execution by registering the actual propagation method with the `ExecutionDelayer` class.

5.3 Response Rating

Since every iServer P2P API request is broadcasted to all members of the groups a peer possibly receives multiple responses. Heinzer [6] proposes to filter and return a selection of responses only. We implemented a rating component where incoming responses are collected and a collection of selected responses can be accessed. The selection is based on rating values that are computed for every response. Any response having a value greater than a given threshold value is added to the collection returned. The rating component contains a main class `RatingManager`, rater classes implementing the `ResponseRater` interface and `Aggregator` implementations. Following we present each of these classes.

5.3.1 RatingManager

A `RatingManager` object is used as the interface between incoming responses and the requesting user. The requesting user calling an iServer P2P API method receives an integer valued ID identifying the broadcasted request. This ID also identifies all responses given to this request. The user retrieves the selected responses by providing this ID.
A response is identified by the request ID and the responding user. For every request this class maintains a set of responses paired with the responding user each. It also manages a set

of registered `ResponseRater` objects that will be used to compute an overall rating value. Multiple rater objects can be registered to rate a response in which case the rating values computed by each rater are aggregated into one overall value. Each rater is associated with a weight which is multiplied with the rating value computed before being aggregated. The weights are automatically normalised, i.e. each divided by the sum of all, such that they add up to 1.0. A previously assigned `Aggregator` object is used to aggregate multiple rating values into one overall value.
When a rating manager is asked to return a collection of selected responses to a particular request, it retrieves the set of responses received so far. For each response it computes and aggregates the rating values returned by every registered response rater. All responses having an overall rating value greater than the given threshold are added to the collection returned.
Figure 5.4 shows the UML definition for the `RatingManager` class. It defines methods to register and unregister `ResponseRater` objects and to set an `Aggregator` object. This class also defines a method `addResponse(Integer, User, Response` which is used by the `IServerResponseHandler` to add incoming responses given by a user responding to a request identified by its ID. The response handler is presented in Sect. 5.4.2. Finally, the requesting user retrieves the selected responses by providing a request ID and a threshold value all returned responses must fulfill.

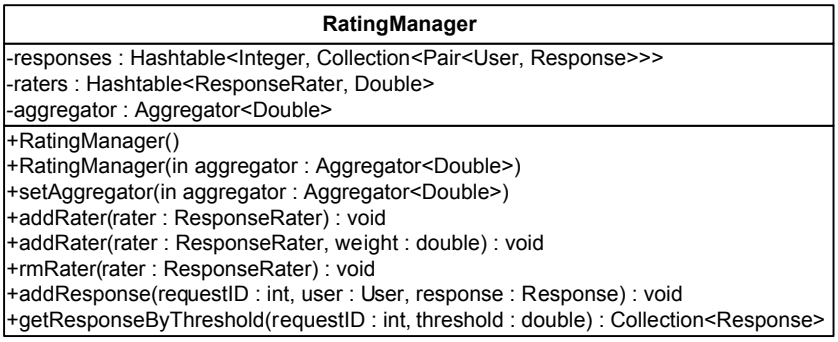

Figure 5.4: UML diagram for `RatingManager` class.

5.3.2 ResponseRater

A rating value for a response is computed by rater classes implementing the `ResponseRater` interface. The interface defines a method that rates a response given by a user responding to a particular request. The information available to the rating method consists of the set of all responses to this request, the responding user and the response to be rated. The method returns a double value ranging between 0 and 1. In Fig. 5.5 we give the UML definition of the `ResponseRater` interface.
We have implemented two response raters:

- A `RateByUser` object accesses the `UserRatingManager` object introduced in the previous Sect. 5.2.2 to retrieve the rating value of the responding user. This value is normalised to range between 0 and 1 before being returned as its own rating value.

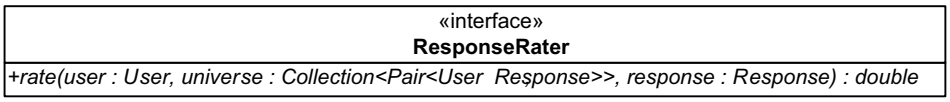

Figure 5.5: UML diagram for `ResponseRater` interface.

- A `RateByFrequency` object returns a rating value proportional to the frequency a response has been given by members of the peer group. If a response has been returned $n_{response}$ times out of n_{total} responses, then the returned rating value is $\frac{n_{response}}{n_{total}}$.

`ResponseRater` objects must be created outside the `RatingManager` class and registered. Since we need to access the `User` and `UserRatingManager` objects they are currently created within the initialisation of the `iServerP2P` object introduced in Sect. 5.4.3.

5.3.3 Aggregator

An `Aggregator<T>` class defines the methods `void addValue(T)` and `T getAggregation()` necessary for aggregation of multiple values. We show its UML definition in Fig. 5.6. We implemented an `Adder` and `Multiplicator` class each extending the `Aggregator<Double>` class. The `Adder` class adds up all added values and the `Multiplicator` class multiplies all values.

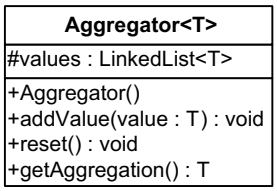

Figure 5.6: UML diagram for `Aggregator` class.

An `Aggregator` object maintains a set of values added which it aggregates according to its specific implementation. Note that this set must be cleared using the `reset()` method whenever an object is to be used to aggregate more than once.

5.4 Peer Service Specific Components and Workflow

To set up an iServer peer service according to our interaction architecture we provide handlers that implemented the JXTA specific requirements enabling the transmission of requests and responses. The iServer specific functionality consists of requesting a remote peer to execute an iServer API method and returning the response. We implemented additional functionality useful in the context of a peer-to-peer application such as online peer retrieval, chat facility

and user rating. We also provide a graphical user interface (GUI) that can be started on demand.
Within the iServer peer service the user plays a more important role than in the iServer web service. Because responses to a request are received from all members of the group, we want to identify peers by the user they represent in order to rate the responses. As in the iServer web service, the user identification can also be used to grant access to iServer data according to the database's user management rules. In this section we present the main components of our iServer peer service implementation.

5.4.1 JXTA

Peer

The `Peer` class encapsulates the functionality of a JXTA peer. When constructed it creates a `Group` object that either retrieves or creates a JXTA group and joins it. A `Peer` object acts as a rendezvous service within the peer group as proposed by Heinzer [6]. A `Peer` object is constructed with a `User` object. Any request or response sent from this object will be enriched with user identification.
The main task of our peer is to register/unregister resolver services. Thus it offers a method taking request and response handler `Class` objects and an identification string uniquely identifying the resolver service the handlers implement. Both handlers are instantiated, the request handler registered with the peer group's resolver service and the response handler registered to be notified on incoming responses by the request handler. Another method taking a service identifying string unrolls the actions performed within the registration process.
This class implements the `Runnable` class and starts a thread containing itself within the constructor. In Fig. 5.7 we present the UML diagram for the `Peer` class.

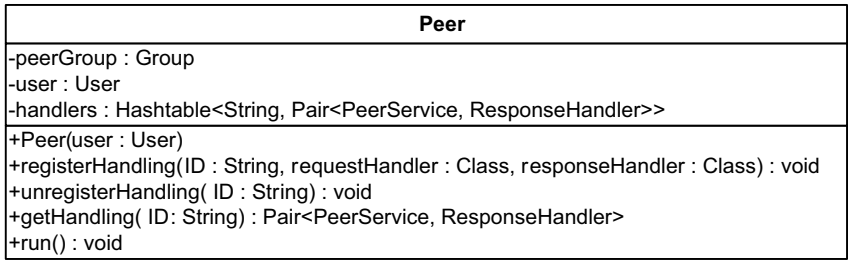

Figure 5.7: UML diagram for `Peer` class.

Group

The `Group` class encapsulates the functionality required to retrieve or create and join a JXTA group. It delegates accesses to the JXTA group such as retrieving of `PeerGroupID`, `PipeService`, `RendezVousService` etc required by the handlers. The group constructed and maintained by this class is an iServer specific group offering the services described above to its members and requiring them to implement and register the
When constructed, it first retrieves the group advertisement which it finds if it has been created and published by peers that are still or have recently been online. If this fails, it tries to open

a file containing the group advertisement. In case the file cannot be found or accessed, the advertisement is created from scratch and saved to a file.
Once the advertisement has been found or created the `Group` object applies for membership providing the authentication credentials. At this moment there is no restriction for membership. The construction is finished once the group has been created and joined. The `Peer` object can then be regarded as being a member of the group since it has access to the group services required to register its services.

ResolverRequestHandler

The `ResolverRequestHandler` is an abstract class extending the `QueryHandler` interface defined by JXTA for registering with the resolver service. It implements the interface methods `processQuery(ResolverQueryMessage)` and `processResponse(ResolverResponseMessage)` that will be called automatically by the resolver service. The `processQuery(ResolverQueryMessage)` method extracts the `Data` object from the query message and delegates the processing to the subclass implementation of the `handleRemoteRequest(HandlingEvent)` method defined in the `RemoteRequestHandler` interface. The `sendQuery(String)` method broadcasts a resolver query message to all other members of the group.
The workflow within a resolver service implementation such as iServer peer service can be described as follows: A `NotifyingLocalRequestHandler` object calls `sendQuery(String)` providing the XML string representation of the `Data` request object as argument. The resolver request handler broadcasts the query within the peer group.
The resolver service notifies all registered `ResolverRequestHandler` objects by invoking their `processQuery(ResolverQueryMessage)` method. The handler delegates the processing of the request to its subclass implementation, e.g. `IServerRequestHandler` which returns an XML string representation of the `Data` response object. The response is sent back to the requesting peer only.
A resolver request handler gets notified by the resolver service when a response arrives. The `processResponse(ResolverResponseMessage)` method handles the response by notifying the response handler registered as listener for incoming responses.
A `ResolverRequestHandler` object is constructed by providing a `User` object identifying the user of the `Peer`. The `User` object will be used by subclasses to automatically enrich a request and response message with a user identification. The `start()` and `stop()` register and unregister the handler object with the peer group resolver service, respectively.
This class extends the `PeerService` class and implements the `RemoteRequestHandler` interface. All request handlers presented in the next section extend this `ResolverRequestHandler` class.

PeerService

This abstract class defines the methods required to register and unregister a service with the peer group resolver service. These methods are `start()` and `stop()`. The `ResolverRequestHandler` class presented above implements this interface.

5.4.2 Handling

In this section we present the implementations of the interaction architecture's handling component specific to peer services and additional service classes. A service is implemented by two classes, a request and response handler. The request handler extending the `ResolverRequestHandler` class is registered with the peer group's resolver service and the response handler is registered with the request handler as a listener for incoming responses.
The handlers introduced in the scope of our interaction architecture were defined with `Data` objects or XML strings representing `Data` objects as parameter and return values. The iServer P2P handlers require more information such as the requesting and responding user and an ID which allows to identify responses corresponding to a particular request. Therefore we implemented a `HandlingEvent` class which contains this additional information along with the `Data` object. The handlers presented in this section take objects of this class as argument.

NotifyingLocalRequestHandler

In Chapt. 2 we presented a request response flow (Fig. 2.1) where the local request handler returns the response with the `handleLocalRequest(Data)` method . In a peer service we receive multiple responses to one request and the time it takes to receive a response suggests the implementation of a listener pattern. Thus we define a new interface `NotifyingLocalRequestHandler` with the `handleLocalRequest(Data)` method returning void. All request handlers presented in this section implement this interface.

ListeningResponseHandler

The counter part of the `NotifyingLocalRequestHandler` interface is a response handler that is notified about an incoming response. Its `handleResponse(HandlingEvent)` method is called whenever a response is received by the resolver request handler. The `ResponseHandler` class presented next implement this interface.

ResponseHandler

This abstract class groups methods and fields common to all response handlers presented in this section. A response handler applies the responses to an object such as an iServer database interface or a user manager. This object is called the `model` and this class defines the corresponding abstract method `setModel(Object)`. When a request response handler pair is registered with a `Peer` object, the registration method creates a `ResponseHandler` object. The object registering a particular service must then set the model of the response handler.
This class defines a `handleResponse(int, User, Message)` method that returns a boolean value. In JXTA multiple responses from one peer may result from one request. In order not to unnecessarily handle multiple responses this method keeps track of responses to a particular request received by a particular user. If it is the first response, the method returns true and it returns false otherwise. Keeping track is accomplished by a `ResolverResponseTracker` object we do not describe any further.
A second private method `streamResponse(Message)` is called by the former to stream the message to a host defined by its name and port number. This can be useful for debugging

purposes and it can be turned on and off by the object constructing an instance of this class. A subclass implementation is free to call `handleResponse(int, User, Message)` to use its functionality. This class extends the `ListeningResponseHandler` class. All response handlers presented in this section extend this class.

iServer Handling

We implemented a request and response handler specific to iServer requests and responses. An `IServerRequestHandler` object takes a `Data` request object, packs it in a `UserMessage` object along with the `User` object representing the user owning the `Peer` object. It broadcasts the XML string representation of this message to all other members of the peer group.
On incoming responses the `IServerResponseHandler` object is notified which handles the response. The response is added to the set of responses managed by the `RatingManager` object set as the model of the response handler.
From an incoming request the `UserMessage` XML string representation is extracted from which the message object is reconstructed. The request handler invokes the `computeResponse()` method on the `Request` object and returns the XML string representation of the `UserResponseMessage` object containing the response and the `User` object representing the user owning the responding peer. Additionally, the response message is enriched with a `Status` object as introduced in Sect. 3.2.

Chat Handling

This service implements a chatting facility for online members of the peer group. So far, a message can be broadcasted to all other members by any peer. The `ChatRequestHandler` class defines the method to broadcast a string message. Its `handleRemoteRequest(HandlingEvent)` extracts the message and notifies its listening response handler. Note that the notification takes place within the request handling as opposed to before when it took place on an incoming response. The handler responds with an acknowledging message.
The response handler has a `ContentModel` object as its model and just appends incoming messages to it. Messages are displayed in a panel contained in the GUI which listens for changes in the `ContentModel` object. Acknowledging messages are streamed if streaming is turned on but ignored otherwise.

Online Peer Retrieval Handling

This service allows the retrieval of `User` objects from responding peers to display a list of users currently online. The `PresenceRequestHandler` request handler has a method `sendRetrievalMessage()` which broadcasts a `UserMessage` object which will be identified as an online peer retrieval message by `PresenceRequestHandler` objects receiving the request. They reply with `UserMessage` messages themselves and the `PresenceResponseHandler` response handler extracts the `User` object from them. It adds the received user representations to its `UserListModel` model object that notifies the GUI panel about changes.

User Validity Handling

User validities management has been presented in Sect. 5.2. The `UserValidityRequstHandler` class defines a method `retrieveValidities()` for requesting validities from all other members in the group. The retrieval is initiated by the vector when it is initialised.
Another `propagateValidities()` method sends a representation of all validities to all other members. This method is invoked by the `UserValidityVector` object when a validity is added or changed.
The request handler defines a field pointing to the `UserValidityVector` object so that it can respond to other peer's requests for validities at any time. The retrieval and propagation methods are delayed using the `ExecutionDelayer` class presented in Sect. 5.2.3.
The request handler either receives a request for sending all validities or it receives a `UserValidityVector` object. Note that a `UserValidityVector` object may be received within a resolver query, i.e. unrequested, or as a response to a request.
In the first case the vector is received in the `handleRemoteRequest(Data)` method defined in the `UserValidityRequestHandler` class. In the second case it is the `processResponse(ResolverResponseMsg)` method defined in the `ResolverRequestHandler` receiving the vector.
However, in both cases the `UserValidityResponseHandler` object is notified and this one treats requested as well as unrequested vectors equally as described in Sect. 5.2.
In case of a request for validities, the request handler replies with a `UserMessage` object containing up to date user validity values.

User Rating Handling

User rating management has been introduced in Sect. 5.2. The handlers are very similar to the ones handling user validity requests and propagation as presented in the previous section.
The `UserRatingRequstHandler` class defines a method `retrieveRatings()` for requesting user ratings from all other members in the group. The retrieval is initiated by the `UserRating` object when it is initialised.
The `propagateRatings()` method propagate a snapshot of all user ratings to all other members. This method is invoked by the `UserRating` object when a user rating is set or altered.
The request handler has a member pointing to a `UserRatingVector` object which always contains a current snapshot of all user ratings. Whenever a rating is set or altered, the `UserRating` object updates this field. Hence, the request handler may respond to other peer's requests for ratings at any time. The retrieval and propagation methods are delayed using the `ExecutionDelayer` class presented in Sect. 5.2.3.
The request handler receives a `UserRatingVector` object either broadcasted by any other peer or as a response to a local request. In the first case the vector is received in the `handleRemoteRequest(Data)` method defined in the `UserRatingsRequestHandler` class. In the second case it is the `processResponse(ResolverResponseMsg)` method defined in the `ResolverRequestHandler` receiving the vector.
In both cases the `UserRatingsResponseHandler` object is notified and this one treats requested as well as unrequested vectors equally as described in Sect. 5.2.
In case user ratings are retrieved by another peer joining the group, the request handler replies

with a `UserMessage` object containing up to date user rating values.

Pipe Communication

We have implemented point-to-point communication using pipes as close to the request response handler pattern as possible. A `JInputPipe` class also extends the `PeerService` class such that it can be loaded using the `registerHandling(String, Class, Class)` method defined by the `Peer` class.
A pipe is created with a pipe advertisement. The `JInputPipe` class tries to read in the advertisement from a file. If this fails, it creates it from scratch and saves it to a file. Once the pipe has been created and is ready to be used, this advertisement can be accessed. It can be sent to another peer, e.g. as part of a broadcasted request, which can then create the output pipe and send messages. A `JInputPipe` object notifies its registered listeners whenever an incoming message has been received.
The counterpart to the `JInputPipe` class is the `JOutputPipe` class. It defines a static method `sendMsg(PipeService, PipeAdvertisement, String)` which creates an instance of itself that will send the message as soon as the output pipe has been created. The `PipeService` object can be retrieved in the `Group` object. The pipe advertisement is created by the `JInputPipe` object where it can be accessed. The third argument is the message to be sent.
At the moment we are not using pipe communication. If it had to be deployed, an appropriate response handler would have to be implemented and registered with the `Peer` object.

5.4.3 iServer P2P

This is the main class of iServer peer service. It creates a `UserManagement` object, starts a `Peer` thread and loads the resolver services. If chosen, the GUI is started and configured. Finally it defines the iServer peer service methods available to the user of the peer service. As in the iServer web service, these methods replicate the methods in the iServer`RequestFactory` class which replicate the methods from the iServer API. These methods form the iServer P2P API and the offered functionality may easily be adjusted in this class. All iServer API methods create their specific `Request` object with the request factory and forward it to the request handler.
An `iServerP2P` object may be started and run in a stand alone fashion. It can also be created within another class. Figure 5.8 depicts the UML diagram for the `iServerP2P` class. The `entitySources(Entity)` method is just an example method for the `iServerP2P` class. The method in squared brackets stands for all iServer P2P API methods that are implemented in this class. All of these methods return an integer value representing the request ID with which corresponding responses can be identified.
The `getResponses(int)` method returns the responses to a given request ID as selected by the rating manager introduced in Sect. 5.3.1. The selection threshold is set as a static member variable in the `iServerP2P`.

5.4.4 Workflow

Figure 5.9 schematically illustrates the interaction in a peer service. The iServer P2P API defines methods to access iServer functionality on a remote peer.

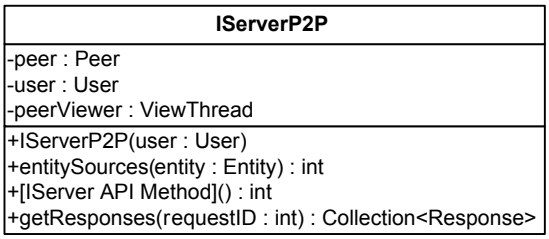

Figure 5.8: UML diagram for `iServerP2P` class.

`Data` objects are used to encode the request which is to be processed remotely and to encode the resulting response. The `Request` object is created by the request factory. The request is handed over to the `IServerRequestHandler` (a) which creates a `UserRequestMessage` object containing it and broadcasts its XML string representation with the resolver service of the peer group (b). The request is received by `RemoteRequestHandler` objects which reconstruct the message and extracts the `Request` object (c). The request is processes which autonomously interacts with the API of the remote database (d) and returns a `Response` object (e). The remote handler uses the `Response` object to create a `UserResponseMessage` object which is returned to the requesting peer as XML string (f). The local handler reconstructs the message and extracts the response (g). The response can either be presented to the user directly or fed back to a local information system (h).
In Fig. 5.10 we illustrate the components participating in request and response handling. The colored boxes reflect the inheritance structure. A blue box represents the complete handling component of a local peer. A bit of a second peer is visible to the right which represents a remote peer. Member fields are written in italic while abstract as well as interface methods are marked with the keyword `abstract`.
The boxes named `[...]RequestHandler` and `[...]ResponseHandler` represent all previously introduced request and response handlers such as `IServerRequestHandler`, `IServerResponseHandler`, `UserRatingRequestHandler` etc.
Handling starts with a local request. For example, an iServer P2P API method calls the method `handleLocalRequest(Data)` providing the `ReflectionQuery` object as parameter. This method invokes `sendQuery(String)` defined in the `ResolverRequestHandler` class. The request is broadcasted to all members of the group with the resolver service. A peer receives a resolver request through its method `processQuery(ResolverQueryMsg`. The request is delegated to the abstract `handleLocalRequest(HandlingEvent)` method which is implemented in the `(...)RequestHandler` classes. This method returns the `Response` object which is sent back to the requesting peer using the resolver service.
A requesting peer receives the response through its `processResponse(ResolverResponseMsg` method. There, a notification of all listening response handlers is initiated. The `notifyListeners()` calls `handleResponse(HandlingEvent)` which is implemented in the `(...)ResponseHandler` classes. The three parameters of the `handleResponse(requestID, User, Message)` method are extracted from the `HandlingEvent` object. This method calls the overriden one in the `ResponseHandler` class which checks for multiple responses using the `ResolverResponseTracker` mem-

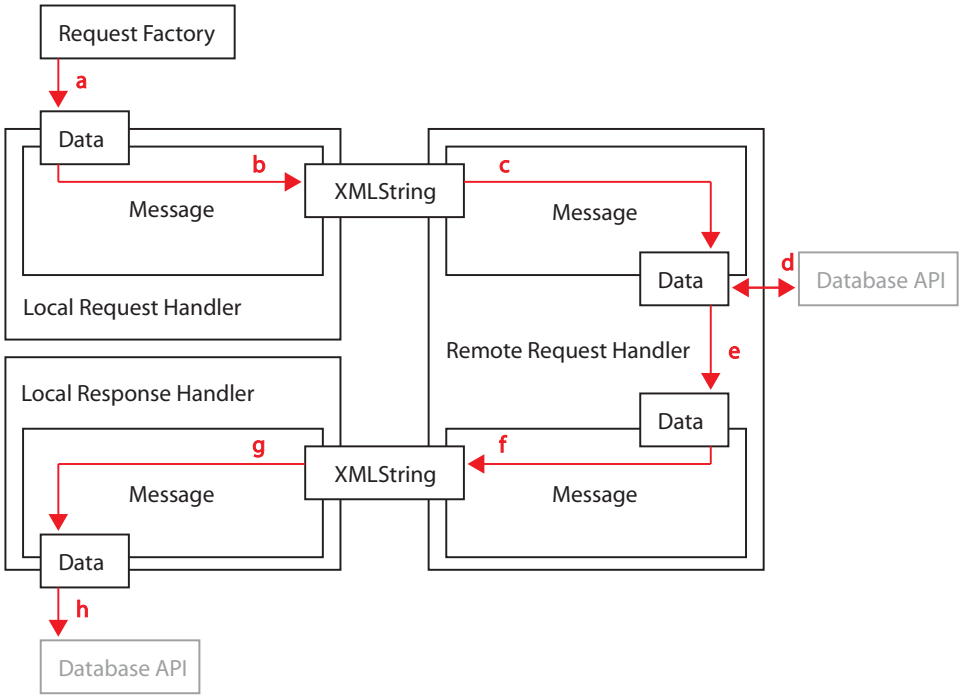

Figure 5.9: Schematic illustration of the interaction between two peers.

ber. If the $(requestID, User)$ tuple is unique, the response is processed with the model set for the response handler.

5.5 Example Usage

In this section we present the usage of our iServer peer service implementation. We give an example iServer P2P API request and show the messages sent between two peers. Further information how to get a peer started and running are given in Appendix A.
Figure 5.11 shows a pseudo code implementation of an iServer P2P API method. All API methods are implemented after the same pattern: in a first step we create the request object with the request factory. Then we get the request handler from the `Peer` object and finally we send the request. The response handler will take care of replies from other peers.
In Fig. 5.12 we show the request message that is broadcasted to all peers in the group. The contained `Data` object is a `OMCollectionReflectionQuery` object as created by the request factory.
Figure 5.13 contains the corresponding response message with which a peer replies to the request. The `Data` member of the message points to a `OMCollectionResponse` object that was created by the `computeResponse()` method of the `OMCollectionReflectionQuery` object.
Figures 5.14 - 5.16 show our GUI in action. The log panel shows all messages sent and received and additional log messages depending on which log level is set to be shown. The existent users panel displays the content of the user validity vector as a list. A double click on

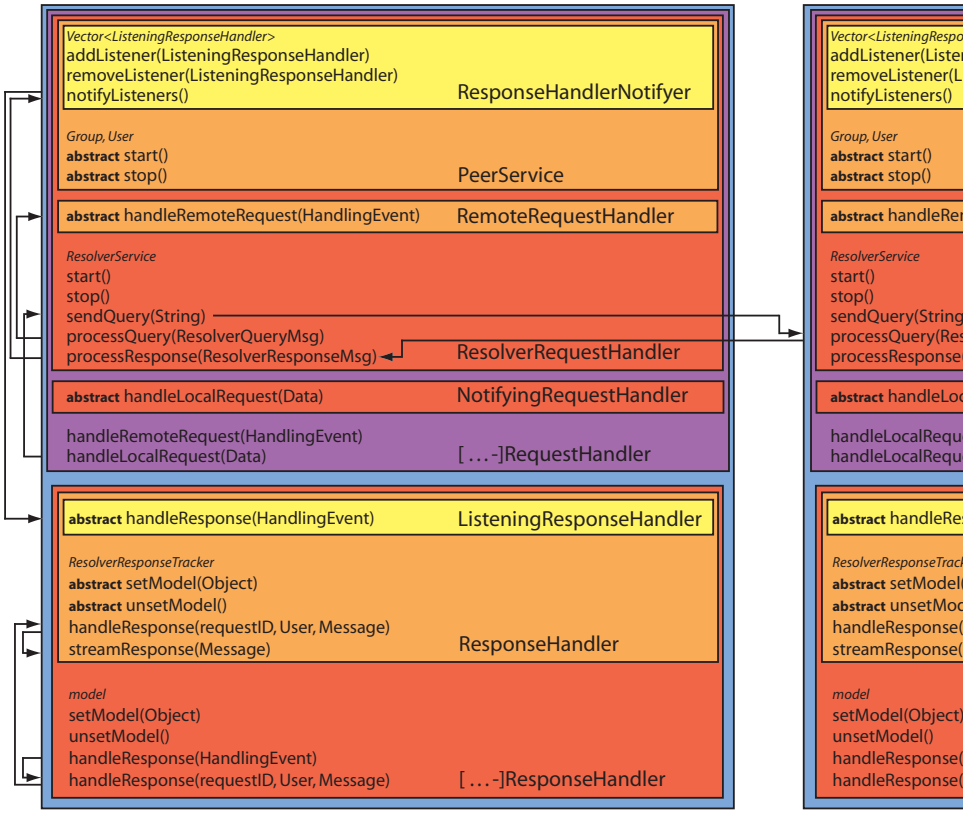

Figure 5.10: Class hierarchy and method calls of request and response handlers.

a user item opens an input dialog where a rating can be typed in. The rightmost panel shows the graph of user ratings.

```
public void entitySources(Entity source)        {

        // get request object at request factory
        Request request =
                RequestFactory.entitySources(source);

        // get request handler
        NotifyingLocalRequestHandler handler =
                peer.getHandling(IServerRequestHandler
                        .HANDLINGNAME).first();

        // broadcast request
        handler.handleLocalRequest(request);

}
```

Figure 5.11: Example peer service method pseudo implementation.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <data data="org.ximtec.iserver.p2p.data.OMCollectionReflectionQuery"
    className="org.ximtec.iserver.core.Entity" methodName="sources">
    <parameterClasses />
    <parameterObjects />
    <instanceObject>
      <omInstance omInstanceName="entity">
        <entity>
          <name>African Savannah</name>
          <creator>
          <authorised />
          <unauthorised />
          <properties />
        </entity>
      </omInstance>
    </instanceObject>
  </data>
  <user user="org.ximtec.iserver.p2p.data.User">
    <string>ombak</string>
  </user>
</message>
```

Figure 5.12: Example user request message sent by a requesting peer.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message>
  <data data="org.ximtec.iserver.p2p.data.OMCollectionResponse">
    <omCollection omTypeName="entity">
      <omInstance omInstanceName="entity">
        <entity>
          <name>Background Buffalo</name>
          <creator>
          <authorised />
          <unauthorised />
          <properties />
        </entity>
      </omInstance>
      <omInstance omInstanceName="entity">
        <entity>
          <name>Background Cheetah</name>
          <creator>
          <authorised />
          <unauthorised />
          <properties />
        </entity>
      </omInstance>
    </omCollection>
  </data>
  <user user="org.ximtec.iserver.p2p.data.User">
    <string>ombak</string>
  </user>
  <status status="org.ximtec.iserver.p2p.data.Status">
    <integer>0</integer>
    <string>request processed successfully</string>
  </status>
</message>
```

Figure 5.13: Example user response message sent by a peer replying on the request.

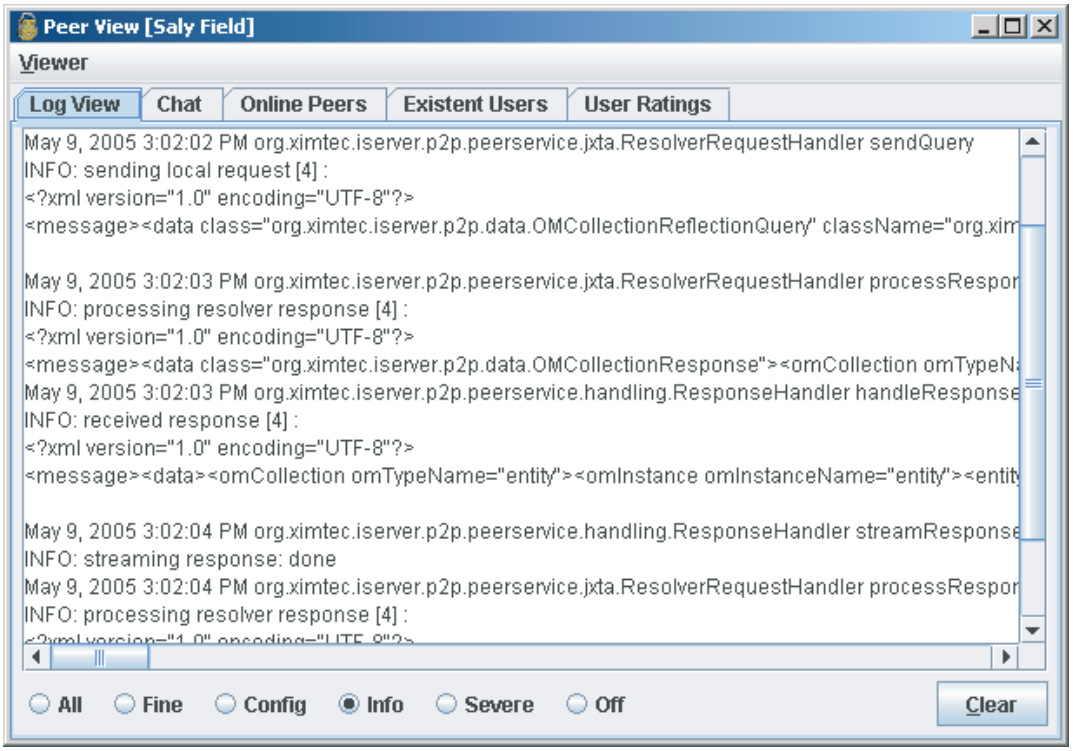

Figure 5.14: Peer GUI, panel for log messages.

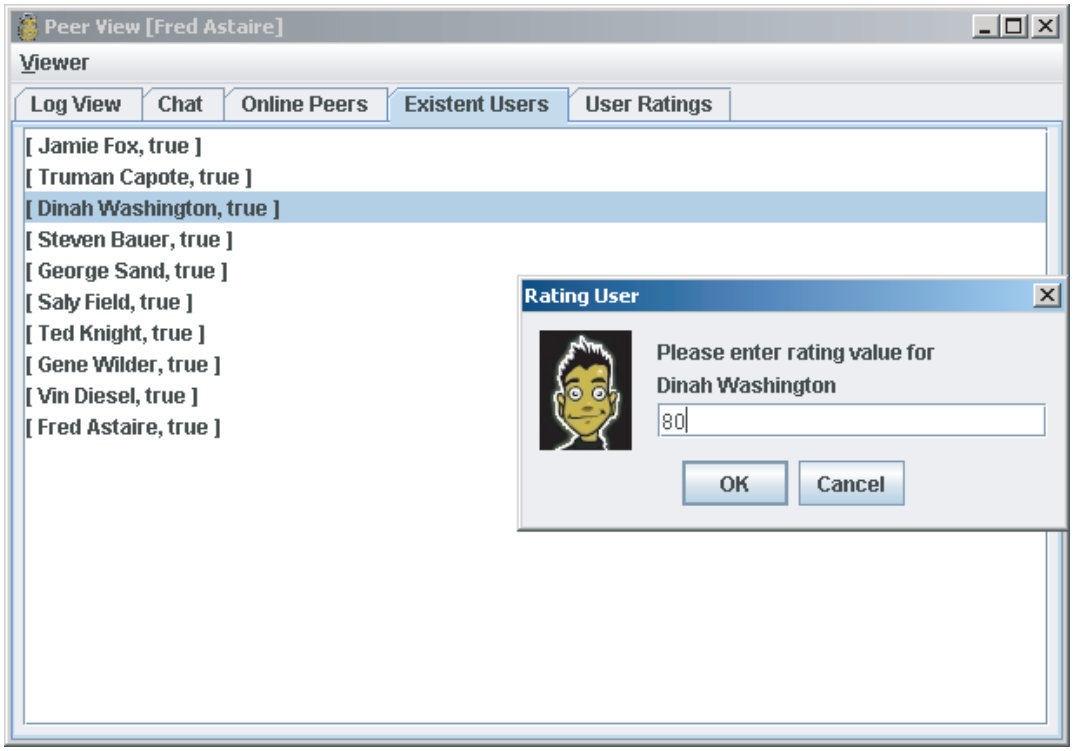

Figure 5.15: Peer GUI, panel showing list of user validities.

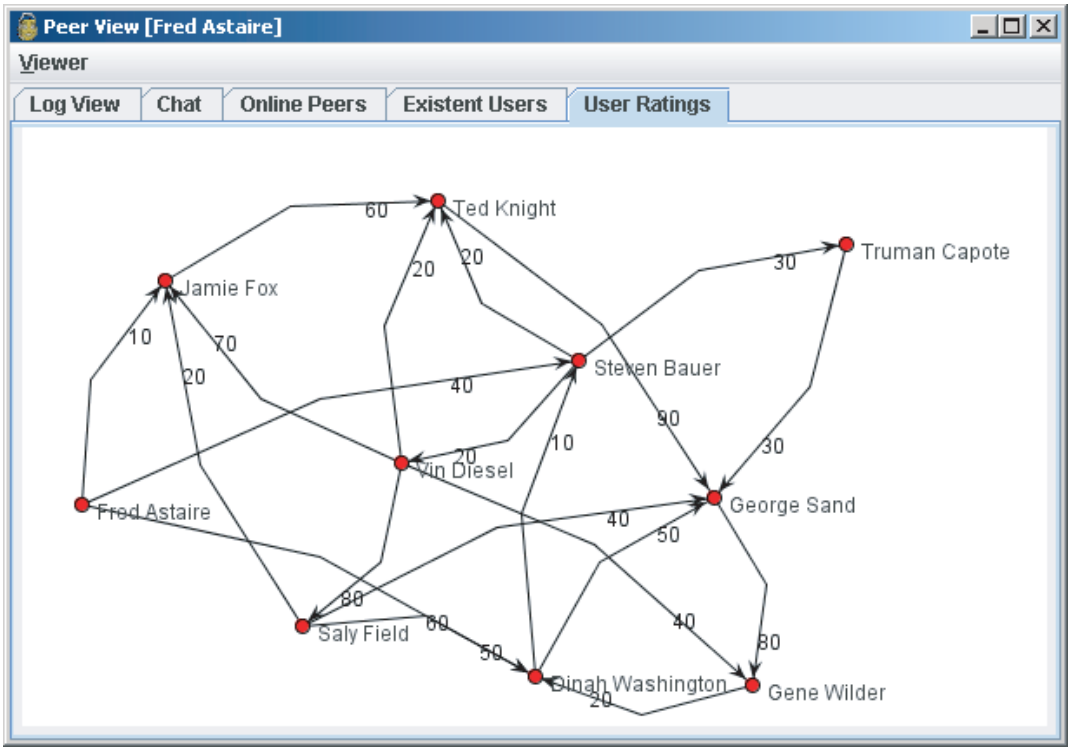

Figure 5.16: Peer GUI, panel visualising the graph of user ratings.

6 Conclusions

6.1 Goals and Results

The goal of this diploma thesis was to design and implement a collaborative iServer information space. The information stored in different iServer instances is to be shared dynamically. The iServer architecture is a platform enabling linking of arbitrarily typed objects. It provides a metamodel defining fundamental link concepts which allow to flexibly structure and link information spaces. The resulting iServer web service and peer-to-peer implementations are based on a generalised interaction architecture. This architecture consists of five components each of them abstracting a particular aspect of interaction in a collaborative information system. Its main achievement is to facilitate the implementation of remote access transparent to the users. In this section we present the main stages of this diploma thesis and discuss the results.

Investigation of iServer, the JXTA P2P framework and the current iServer P2P prototype. With the help of Signer's [14] publication we quickly familiarised with the iServer architecture, its concepts and philosophy. We found good introductory literature [16] as well as helpful technical manuals on JXTA technology [11]. Heinzer [6] had previously compared various peer-to-peer technologies which allowed us to focus on JXTA from the beginning.
As a result of this stage we designed and implemented a generalised architecture for distributed information systems. After analysing the most commonly used remote interaction frameworks such as web service and peer-to-peer we identified five independent components that make up interaction between participants of a distributed information system. Each of these components can be extended or altered independently which makes our architecture flexible and adjustable to specific application requirements.

Implementation of an iServer web service interface based on the available iServer XML object representation. Definition of a web service API and a query interface for iServer. Due to the results from the previous stage and a short period of familiarisation with web ser-

vice concepts we were quickly able to set up an iServer web service. The interface defines a web service method for each iServer API method. Such a method takes the required parameters as XML string representations conforming to the iServer XML schema definitions. It returns the result wrapped with a message that also includes information about the processing of the request, i.e. an error code and message. The web service class starts up the database to be used automatically. All preferences — mainly the name and location of the database — can be set in a configuration file.
Our web service implementation fully benefits from the flexibility of the interaction architecture designed in the previous stage. The service functionality offered to a client, the encoding of parameter objects and returned messages as well as the content of a response message can be adjusted easily. The web service can also be integrated into a user management system since our architecture defines classes to transmit user information contained in objects of any type.

Implementation of a distributed iServer version based on findings from an earlier iServer P2P prototype. We implemented an iServer P2P system based on our interaction architecture. The components implemented for the web service could be used with little adaptations only. Due to the flexibility of the components they could easily be integrated into the JXTA framework. Nevertheless the development of the peer-to-peer system turned out to be a complex and challenging undertaking. Multiple redesigns were inevitable but the partitioning of the functionality into independent components as well as exhaustive unit testing helped to accept the challenge.
The JXTA framework proved to be a very powerful framework facilitating the implementation of a peer-to-peer application as much as possible. Due to the restrictions of console input and output we decided to implement a graphical user interface for debug purposes. This interface evolved to an iServer P2P managing system while still being an optional feature.

Development of a component for the rating and filtering of information exchanged in the collaborative iServer environment. In a community of iServer instances a request potentially generates multiple responses. Nevertheless not all of the responses are equally valuable. Experiences with other peer-to-peer networks such as file sharing have revealed some problems associated with decentralised and democratic publishing and consuming of content. Some members of the community tend to consume without publishing while others publish unsolicited or malicious content. These problems have been addressed in the previous iServer P2P prototype implemented by Heinzer [6] by introducing a filtering of responses. We have implemented a user rating management where ratings are propagated across the community. Each member stores a local snapshot of all currently exiting ratings which allows him to transitively follow sequences of ratings. Thus a member can infer the trustworthiness of another member he has not explicitly rated.
We have implemented a response filtering based on user rating, frequency analysis and arbitrary criteria that can be expressed as a function of the set of responses received each coupled with the responding user. The rating architecture has mainly been taken over from Heinzer's work where it had been designed to enable a wide variety of filtering methods. Once initialised the filtering is achieved transparently to the user.

6.2 Future Work

Collaborative information sharing currently gives rise to new approaches in the area of information systems. Taking our interaction architecture and peer-to-peer technologies as a starting point, we propose to broaden the concepts developed in this work to support a wider range of information systems. The architecture proved to be a functional abstraction of collaborative interaction within iServer P2P. We believe that the application to other platforms would help to further improve its generalisation ability. Components may have to be added, removed or altered to increase flexibility and extendibility.
Decentralised and democratic information sharing as suggested by a peer-to-peer architecture raises questions some of which have been resolved superficially in the scope of this work. Problems such as object identity across database instances implementing a common schema, object transfer between database instances with differing schemas as well as the automatic filtering based on user and object content rating must be addressed more exhaustively.
We also suggest to investigate new application areas where collaborative information sharing can profit from acquisitions in mobile and ubiquitous computing and, vice versa, current technologic developments can be enriched with collaboration facilities.

A
User's Manual

In this section we give the information necessary to run iServer web and peer service. Further guidance about the employment of specific classes can be found in the Javadoc documentation and unit test classes.

A.1 iServer Web Service

For setting up an iServer web service we recommend to follow the steps five till nine in the web service tutorial appended in Appendix B. The following packages must be accessible to the deployed service:

- `org.sigtec.iserver.iserver.p2p.architecture`
- `org.sigtec.iserver.iserver.p2p.data`
- `org.sigtec.iserver.iserver.p2p.jdom`
- `org.sigtec.iserver.iserver.p2p.messaging`
- `org.sigtec.iserver.iserver.p2p.resource`
- `org.sigtec.iserver.iserver.p2p.webservice`

A `javax.xml.transform.TransformerFactoryConfigurationError` currently causes an exception with JDK 1.5. We found a fix for this problem in the Sun Developer Network [15]: The `xml-apis.jar` file must be removed from the `Tomcat/common/endorsed/` directory.
The web service starts the iServer database independently, thus its name and location must be set in the configuration file `config.properties`. All configuration files can be found in `Tomcat/bin/`. If any configuration file does not exist it will be created on the first start up

and filled with default settings. If it exists, the preferences are read from it. All other settings besides the name and location of the database do not need to be adjusted for regular usage. In Fig. A.1 we present Java code that consumes an iServer web service API method. The method takes no parameters, thus it can be executed without further preparations.

```
// example iServer web service URL
URL iServerURL =
        new URL(
                "http://localhost:8081/iServerWebService/
                        services/iServerWebService"
        );

// get the web service
IServerWebServiceService iServerService =
        new IServerWebServiceServiceLocator();
IServerWebService iServer =
        iServerService.getIServerWebService(iServerURL);

// consume a web service method
String response = iServer.iServerCollectionEntities();
```

Figure A.1: Java code that consumes an iServer web service requiring no parameters.

In Fig. A.2 the consumed web service method requires one parameter of type `Entity`. The parameter object `instance` is prepared by interacting with the example iServer database. It is then transformed into its XML string representation.

A.2 iServer Peer Service

The main class for starting up an iServer peer is `iServerP2P`. It defines a `main(String[])` method that creates and runs an `iServerP2P` object. Following is a list of its arguments:

- User name (e.g. "Fred Astaire"): A `User` object currently consists of a string which is given as argument.
- Display GUI (e.g. true): $\in (false, true)$. Starts the GUI if true.
- Stream to host (e.g. 127.0.0.1): Host to which messages received and log messages are streamed.
- Stream to port (e.g. 2781): See *Stream to host* argument.
- Receive stream on port (e.g. 2781): Where a server listens for streamed messages and log messages to be displayed.

The first argument is mandatory. All other arguments can be omitted (all of them or none of them) in which case no GUI is shown.
An object of this class can also be created using its constructor. The constructor takes the same arguments as the main method.
All preferences that can be set by the user are stored in configuration files named `config.properties` and `configP2P.properties`. If any of these files does not exist it will be created on the first start up and filled with default settings. If they exist the preferences are read from them. Most importantly, the iServer database name and location must be set according to the database to be used. No other settings must be adjusted for regular usage. Most settings must be adjusted carefully since all peers must have them in common in order to be able to communicate properly.
The user validities and ratings are stored in a file created on the first start up and maintained to always contain the validities and ratings currently shared by all peers. Note that if these files are edited while the iServer peer service is running, the changes will not be propagated and they will be lost as soon as the local peer receives a validity/rating update from any other member of the group. The service must be stopped while editing and restarted afterward.
The creation of a `iServerP2P` object will automatically start JXTA. During the first startup a window is shown where preferences can be set. Figures A.3 and A.4 show two panels contained in this window where something has to be set. A red asterisk denotes a field that has to be configured in addition to the default settings. In case a peer does not seem to communicate with other members of the group the `.jxta` folder may be erased and the peer restarted.
The `iServerP2P` class features the iServer peer service methods that can be called once the object is initialised. Each of these methods returns an integer value identifying the request that has been sent. A selection of the responses received can be retrieved with the method `getResponses(int)` where the argument identifies the request to which the responses are to be returned.
Figure A.5 shows the Java code starting up an `iServerP2P` object with its GUI. The object is kept alive until the user kills it. No requests are sent but incoming requests are processed.

In Fig. A.6 we present the code necessary to start up an iServer peer that broadcasts a request to all other members of the group and accesses a selection of the responses. The GUI is started as well.

```
// example iServer DB startup
String applicationName = "naturalHistory";
String applicationRoot = "D:/Develop/db";

Database.setApplicationName(applicationName);
Database.setApplicationRoot(applicationRoot);
Database.getProperties();

// an example Entity object
String entityName = "African Savannah";
OMCollection collection =
        IServer.collectionEntities().select(
                "name", "=", entityName);
OMInstance instance = collection.getFirstInstance();

// example iServerURL
URL iServerURL =
        new URL(
                "http://localhost:8081/iServerWebService/
                        services/iServerWebService"
        );

// get the web service
IServerWebServiceService iServerService =
        new iServerWebServiceServiceLocator();
IServerWebService iServer =
        iServerService.getIServerWebService(iServerURL);

// create XML string representation of parameter
Document document =
        new Document(XMLElementFactory.toXML(instance));
XMLOutputter out =
        new XMLOutputter();
String request = out.outputString(document);

// consume web service
String response = iServer.entitySources(request);
```

Figure A.2: Java code that consumes an iServer web service requiring an `Entity` object as parameter.

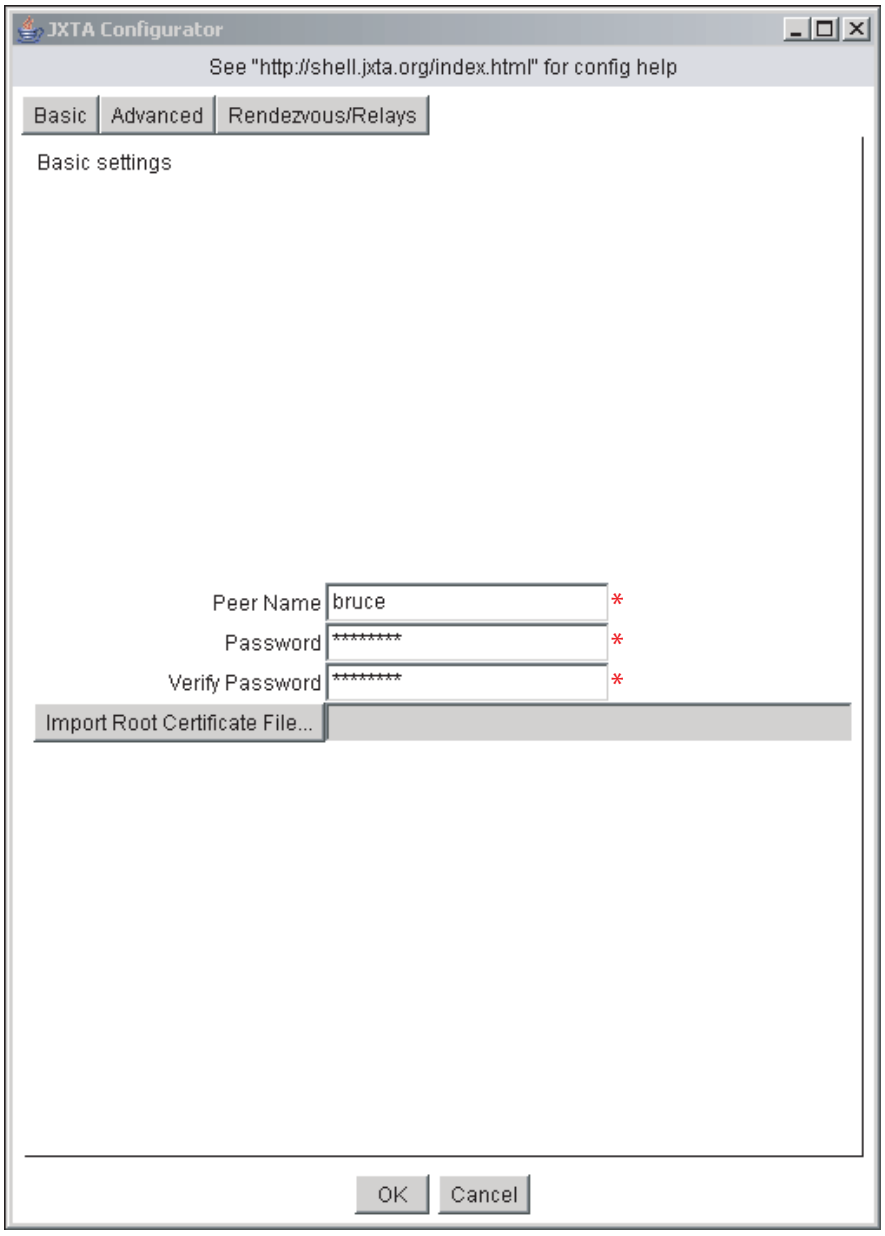

Figure A.3: Configuration user interface of JXTA. The fields marked with a red asterisk must be set in addition to the default settings.

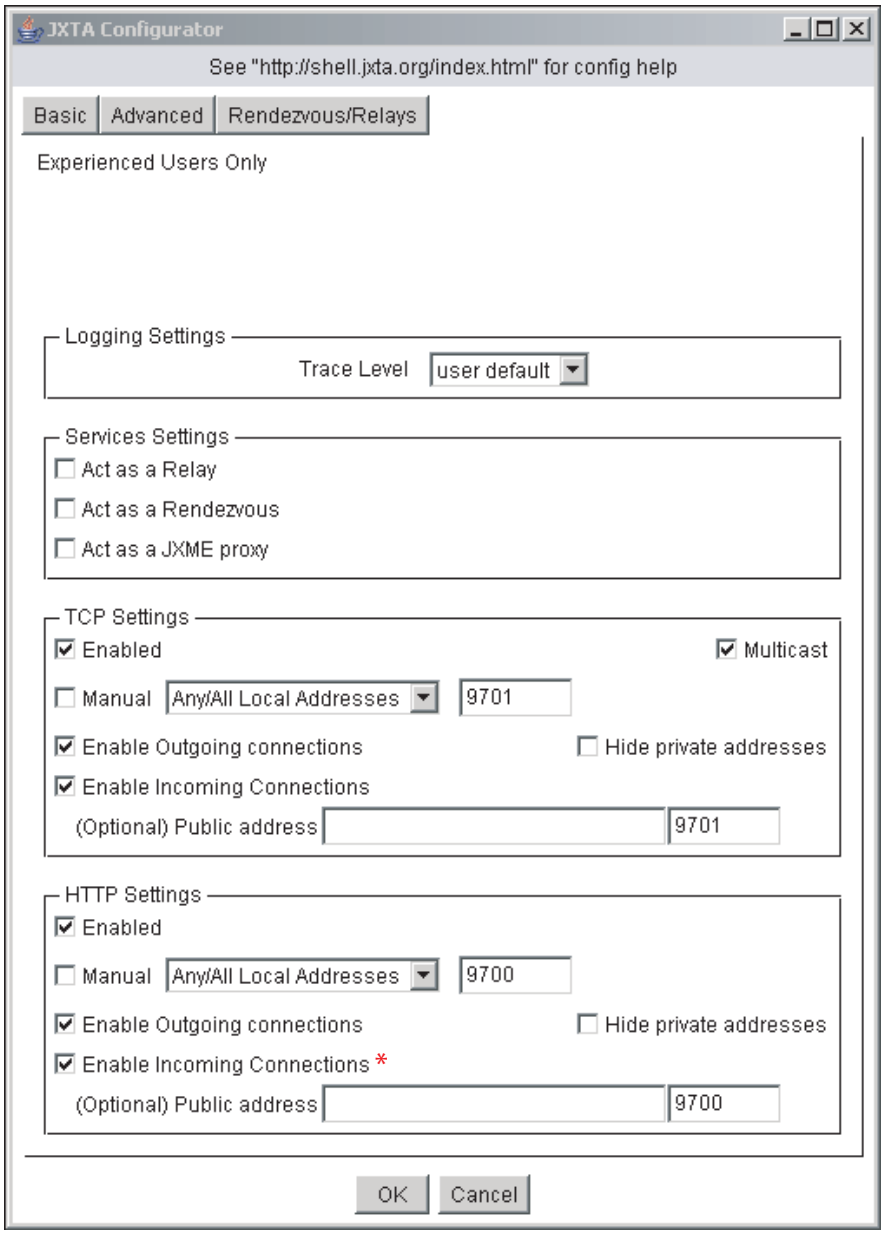

Figure A.4: Configuration user interface of JXTA. The box marked with a red asterisk must be checked in addition to the default settings.

```
// example iServer DB startup
String applicationName = "naturalHistory";
String applicationRoot = "D:/Develop/db";

Database.setApplicationName(applicationName);
Database.setApplicationRoot(applicationRoot);
Database.getProperties();

// create a user
User user = new User("Fred Astaire");

try {

        // create peer, group, register handlers etc...
        IServerP2P peer = new IServerP2P(user, true,
                IServerP2P.LOGTOHOST, IServerP2P.LOGTOPORT,
                IServerP2P.LOGFROMPORT);

        // ... and wait
        while (true) {
                Thread.sleep(2000);
        }

} catch (Exception e) {
        e.printStackTrace());
}
```

Figure A.5: Java code to start an iServer peer service waiting to interact with other peers.

```
// example iServer DB startup
String applicationName = "naturalHistory";
String applicationRoot = "D:/Develop/db";

Database.setApplicationName(applicationName);
Database.setApplicationRoot(applicationRoot);
Database.getProperties();

// create a user
User user = new User("Fred Astaire");

// an example Entity object
String entityName = "African Savannah";
OMCollection collection =
        IServer.collectionEntities().select(
                "name", "=", entityName);
OMInstance instance = collection.getFirstInstance();


try {
        // create peer, group, register handlers etc...
        IServerP2P peer = new IServerP2P(user, true,
                IServerP2P.LOGTOHOST, IServerP2P.LOGTOPORT,
                IServerP2P.LOGFROMPORT);

        // ..., request ...
        int requestID = peer.entitySources((Entity) instance);

        // ... get responses ...
        Collection<Response> responses =
                peer.getResponses(requestID);

        // ... and wait
        while (true) {
                Thread.sleep(2000);
        }

} catch (Exception e) {
        e.printStackTrace());
}
```

Figure A.6: Java code to start an iServer peer service that requests another peer's iServer database.

B
Web Service Tutorial

In this chapter we put a tutorial for web services that was written by Michael Grossniklaus in HTML format.

B.1 Translator Web Service

This website describes the **Translator Web Service** developed at the Global Information Systems Group at the Swiss Federal Institute of Technology. This Web Service was implemented for educational purposes only and cannot be used commercially. On this website you can download the entire **Translator Web Service** including two clients.

B.1.1 Overview

The **Translator Web Service** provides a simple interface that allows translating strings from one language into another. The exact description of the interface can be found in the Web Service's wsdl file.

B.1.2 Demo

To get a notion of the capabilities of the **Translator Web Service** you may download a demo application. Save translator.jar to your hard disk and unzip it. This will create a new directory *webservice* in which you will find start-up scripts for Windows and UNIX. Change to the *webservice* directory and execute either *translator.bat* or *translator.sh*.

When everything is working you should see the window shown in Fig. B.1 appear on your screen.
Congratulations, you have successfully installed your **Translator Web Service** client. If you have problems check that the following requirements are satisfied on your system.

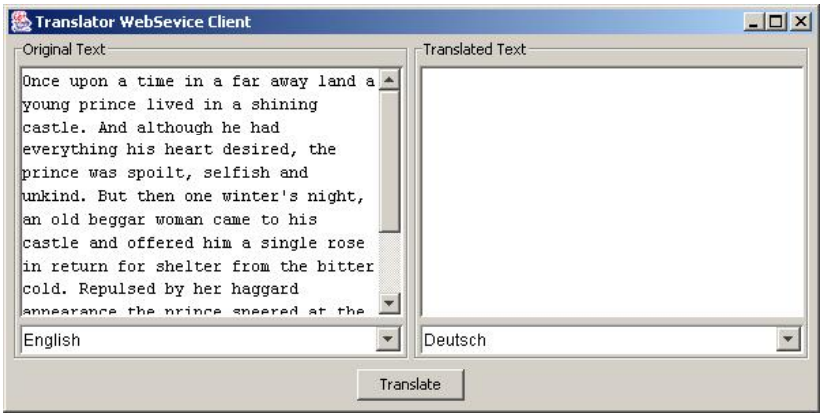

Figure B.1: Translator user interface.

- Do you have a recent version (at least version 1.3.0) of the Java Runtime Environment installed on your system?
- Did you add the path to the java binary to your path environment variable? This can easily be checked by opening a command prompt and typing ”java”. If the system tells you that it cannot find the requested application, modify your path environment variable.
- The websevice/lib directory should contain the newest version of Apache Axis, Apache Xerces.

B.1.3 Instructions to run it on your Machine

If you are interested in the Translator Web Service and running Web Services in general read Sect. B.2 containing detailed instructions on how to implement and build this Web Service.

B.2 Implementing the Translator Web Service with Apache Axis

This page takes you through the steps necessary to implement the **Translator Web Service** with Apache Axis 1.1 and publish it on your Apache Tomcat 4.x application server. This is no tutorial on how to install Apache Tomcat or Apache Axis. Make sure that you have installed these packages and that they are working properly. Refer to the respective tutorials and installation manuals if you are not sure whether you meet these prerequisites.

B.2.1 Step One: Implement a Java class

Each Web Service is described by a wsdl file which contains the Web Service’s interface in XML. Therefore, when implementing a Web Service from scratch it is important to focus on the interface the service should offer. From this interface a wsdl file has to be generated. Of course, this could be done by hand, but the Apache Axis distribution provides a much more elegant way.

Instead of writing wsdl files by hand, we will use the *Java2WSDL* tool from Apache Axis. To use this tool however we first must implement a small Java class or interface with all methods that should be available on the Web Service.
The necessary class for the **Translator Web Service** example is displayed in Fig. B.2.
Copy the source code into a file named *Translator.java* and compile it using the *javac* binary which is part of the Java Development Kit (JDK). If you manage to complete this procedure without any errors you have successfully completed the first step in creating your own **Translator Web Service**.

```
package ch.ethz.globis.demo.translator;

public interface Translator {

    /**
     * Returns the given text string translated according
     * to the specified language pair.
     * @param text
     *            original text
     * @param sLang
     *            source language of the original text
     * @param tLang
     *            target language of the returned translation
     * @return    translated text
     */
    public String getTranslation(String text, String sLang,
            String tLang) throws Exception;

}
```

Figure B.2: Necessary class for the **Translator Web Service** example.

B.2.2 Step Two: Generate a WSDL File

The next step is to generate a wsdl file from the above Java class. As already mentioned there is a tool called *Java2WSDL* included in the Apache Axis distribution which does exactly that. The tool is located in the *org.apache.axis.wsdl* package.
In Fig. B.3 the call to *Java2WSDL* to generate the wsdl file for our Translator Web Service example is illustrated. The command is split onto several lines for convenience only. To execute *Java2WSDL* just type it as one line!
If you have done everything right up to now, the above command generates the appropriate wsdl file. If not, here are some hints and explanations what is going on and may be wrong!

- The most common mistake when working with Java is a wrong class path. The classpath in the above example is set explicitly to the current directory and some jar files

```
java -cp ".;axis.jar;commons-discovery.jar;
                commons-logging.jar;jaxrpc.jar;
                log4j-1.2.4.jar;saaj.jar;wsdl4j.jar;"
      org.apache.axis.wsdl.Java2WSDL
      -n "urn:Translator"
      -o "Translator.wsdl"
      -l "http://localhost:8080/translator/
                services/Translator"
      -p "ch.ethz.globis.demo.translator"
                "urn:Translator"
      -s "Translator"
      ch.ethz.globis.demo.translator.Translator
```

Figure B.3: Call to *Java2WSDL* to generate the wsdl file for our **Translator Web Service** example.

containing the required libraries. Make sure you have all these files and that their path is correctly included into the classpath.

- If you changed the package name of the class given in the first step, then you should change the value of the p parameter as well as the class included on the last line of the above example to reflect the changes you've made.
- Should you feel uncertain about the values given to the parameters of *Java2WSDL* above, please consult the documentation of *Java2WSDL* included in the Apache Axis user's guide.

B.2.3 Step Three: Generate Client and Server Java Classes

Next, we want to implement a Web Service and a client application, that communicate using the interface defined in step one. As we are already in possession of a wsdl file specifying this interface, we can again let Apache Axis do some work. Included in the distribution is a tool called *WSDL2Java* that takes a wsdl file and generates the following.

- Server side bindings
- Client side stubs
- Deployment files to deploy and undeploy the Web Service on Apache Axis

As with *Java2WSDL* before, the call to *WSDL2Java* is quite complex. To make it easier for you the version as required by our example is illustrated in Fig. B.4.
Again, if there should be any problem check if the points given in the section before are correct. If so the tool will create the following files

- *Translator.java*: This file contains a new version of the interface defined by us in step one. It replaces the before implemented interface, but instead of a simple copy, this version also includes the appropriate uses of the *java.rmi* package.

```
java -cp ".;axis.jar;commons-discovery.jar;
                commons-logging.jar; jaxrpc.jar;
                log4j-1.2.4.jar;saaj.jar;wsdl4j.jar;"
      org.apache.axis.wsdl.WSDL2Java
      -s
      -S
      -o "."
      -d Session
      -N "urn:Translator" "ch.ethz.globis.demo.translator"
      Translator.wsdl
```

Figure B.4: Call to *WSDL2Java*.

- *TranslatorService.java*: Abstract representation of the the Web Service. This Java interface provides methods to get a *Translator* object from the Web Service that complies to the specification according to the *Translator.java* interface.
- *TranslatorServiceLocator.java*: Concrete implementation of the *TranslatorService* interface. This class stores the default location of the Web Service and provides utility methods to query the Web Service for ports based on service endpoint interfaces.
- *TranslatorSoapBindingStub.java*: Client side implementation of the Web Service interface containing all necesserary calls to establish remote communication. The stub implements interface *TranslatorPortType*.
- *TranslatorSoapBindingSkeleton.java*: Server side skeleton class that is deployed onto the Apache Tomcat application server. This class offers the methods specified in our original interface.
- *TranslatorSoapBindingImpl.java*: Actual implementation of the Web Service's functionality on the server side. In this file we will have to insert our implementation in the next step.
- *deploy.wsdd*: Deployment descriptor for our Web Service containing the appropriate classes and namespaces.
- *undeploy.wsdd*: Undeployment descriptor to remove the Web Service from Apache Axis.

Note: Before executing *WSDL2Java* it is useful to backup and remove the original source of class *Translator.java*. Otherwise it won't be generated by the tool.

B.2.4 Step Four: Fill in the Blanks

Two things remain to be implemented. First we'll have to change the source code of class *TranslatorSoapBindingImpl.java*. Although the *WSDL2Java* tool generated a correct class that can be compiled, it didn't include the functionality as imagined by us. Consequently

the source code of the generated class simply provides the default implementation shown in Fig. B.5.

```
package ch.ethz.globis.demo.translator;

import java.rmi.RemoteException;

public class TranslatorSoapBindingImpl implements Translator {

    public String getTranslation(String arg0, String arg1,
            String arg2) throws RemoteException {
        return null;
    }

}
```

Figure B.5: Source code of the generated *TranslatorSoapBindingImpl* class.

For reasons of compactness we will not display the resulting implementation of class *TranslatorSoapBindingImpl* in full detail. The complete source code of the **Translator Web Service** can be downloaded as a jar file. Please refer to the source code in this directory for further details.

B.2.5 Step Five: Write a Client

To really see if the Web Service is working we will be needing a client that connects to the Web Service and sends some queries. The simplest user interface is always a command line based approach. The class included in Fig. refcode:Client implements a rudimentary text based client for the **Translator Web Service**.
Note: When compiling the client, be sure to include the same jar files as above and the directory where the client source code is located in your classpath.

B.2.6 Step Six: Setup Apache Tomcat 5.x

Before we can deploy the Web Service using the Apache Axis tools and the deployment descriptors that were generated in step three, we must setup our Apache Tomcat 4.x application server to host and run the **Translator Web Service**.
Note: In the following sections we will assume, that *CATALINA_HOME* is an environment variable set to the directory where you installed your Apache Tomcat 4.x server.
The first step in configuring the Apache Tomcat 4.x application server is to create an appropriate directory where the code and setup of the Web Service will be stored. To do so it is necessary to create the directory structure shown in Fig. B.7 as a subdirectory of *CATALINA_HOME/webapps*. The classes directory will afterward contain the compiled Java classes of our Web Service that we implemented in the steps before. Directory *lib* will be used to store all required libraries as jar files.

```
package ch.ethz.globis.demo.translator;

public class TranslatorTextClient {

   public static void main(String[] args) {
      try {
         TranslatorService translatorService =
               new TranslatorServiceLocator();
         Translator translator = translatorService.
                getTranslator();
         String text =
               translator.getTranslation(args[0], args[1],
                        args[2]);
         System.out.println("Translation: " + text);
      } catch (Exception e) {
         System.out.println(e.getMessage());
         e.printStackTrace();
      }
   }

}
```

Figure B.6: A rudimentary text based client for the **Translator Web Service**.

The second step in setting up Apache Tomcat 4.x is to copy the classes and libraries necessary to run the **Translator Web Service** on the application server to the appropriate directories. The following list is a summary of the steps you need to execute to copy your code onto Tomcat.

- Copy the *ch* (or whatever the root of your package hierarchy is) directory structure containing all your compiled classes to the *classes* directory.
- Copy *axis.jar*, *commons-discovery.jar*, *commons-logging.jar* and *wsdl4j.jar* to the *CATALINA_HOME/shared/lib* directory on the application server.
- Copy *log4j-1.2.4.jar* to the *CATALINA_HOME/common/lib* directory on the application

Figure B.7: Directory structure for **Translator Web Service**.

server.

- Copy *jaxrpc.jar* and *saaj.jar* to the *CATALINA_HOME/common/endorsed* directory on the application server.

Note: These steps apply to a first time setup of the Web Service on the application server. Once you have copied all these classes to the server directories, you will only need to re-copy those classes that you change.
Note: Whenever you copy new code to your Apache Tomcat 4.x application server, it is advisable to restart the server to make sure that the new classes are used.

B.2.7 Step Seven: Setup Apache Axis

On the Apache Tomcat 4.x application server there has to be a servlet that will be invoked by Tomcat on incoming requests to the Web Service. This servlet will communicate with the client via the SOAP protocol on behalf of our Web Service. The Apache Axis distribution contains such a servlet called AxisServlet that acts as a SOAP wrapper for arbitrary Java classes. The only thing we have to do is to tell Tomcat that it should use this servlet for our Web Service. To do so, we put the file *web.xml* (Fig. B.8) into the *WEB-INF* in the Web Service's server directory.
Note: If you are familiar with Java Servlets and Apache Tomcat you should recognize this procedure, as it is nothing related to Web Services specifically, but simply the general procedure of installing a servlet on an application server.
You can already check if you have done everything correct up to now. Start your Apache Tomcat 4.x server and open the URL http://localhost:8080/translator/services/AdminService in your webbrowser. Among other information, you should see the message "*Hi there, this is an AXIS service!*".
Note: If you don't get to this page, try restarting your Apache Tomcat application server after modifying the file *web.xml*.

B.2.8 Step Eight: Deploy the Translator Web Service

Right now we have Tomcat and Axis running and the whole **Translator Web Service** code is on the application server. The only missing part is the link between Apache Axis and our code. This link is important because in the end, the *AxisServlet* has to know to which classes it has to forward the request. This configuration information is stored in the *WEB-INF* directory in an xml file called *server-config.wsdd*.
Again it would be possible to create and edit this file by hand, but Apache Axis provides us with a much more elegant and automatic way of creating this file. The *AdminClient* tool included in the Apache Axis distribution uses the deployment descriptors created in step three to deploy a Web Service at a given URL. For our example the appropriate command is given in Fig. B.9.
When typing the above command, you should see the output shown in Fig. B.10 in your command prompt.
Note: If you don't succeed to deploy the Web Service using the *AdminClient*, make sure that you are able to access the URL http://localhost:8080/translator/services/AdminService in your webbrowser. If you see an exception when requesting this page, make sure, that

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
<web-app>
  <display-name>Apache-Axis</display-name>
  <servlet>
    <servlet-name>AxisServlet</servlet-name>
    <display-name>Apache-Axis Servlet</display-name>
    <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>AdminServlet</servlet-name>
    <display-name>Axis Admin Servlet</display-name>
    <servlet-class>org.apache.axis.transport.http.AdminServlet</servlet-class>
    <load-on-startup>100</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/servlet/AxisServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>*.jws</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>AdminServlet</servlet-name>
    <url-pattern>/servlet/AdminServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Figure B.8: File *web.xml*.

you have copied all necessary libraries to the directories described above. For some peculiar mechanisms in the Tomcat classloader Axis does not work, if you copy the jar files to the Web Service's *lib* directory instead of using the global *CATALINA_HOME/common/lib* directory!
Note: Again, now would be a good point in time to restart your Apache Tomcat 4.x application server.

B.2.9 Step Nine: Give it a Test Drive!

Finally after all the work the Web Service is up and running and ready for a test drive. To see if everything was set up correctly we use the client implemented in step five. After it is compiled, we can invoke the client as shown in Fig. B.11 with three arguments. The first argument is the text we want to translate, the second is the source language of this text and the third parameter is the target language into which the **Translator Web Service** should translate the text.
After having typed the above command at the command prompt, the output shown in Fog.B.12 should appear on the console window.
Congratulations! You have successfully implemented and installed your own version of the

```
java -cp ".;axis.jar;commons-discovery.jar;
                commons-logging.jar;jaxrpc.jar;
                log4j-1.2.4.jar;saaj.jar;wsdl4j.jar;"
        org.apache.axis.client.AdminClient
        deploy.wsdd
        -l "http://localhost:8080/translator/
                services/AdminService"
```

Figure B.9: Command to deploy our **Translator Web Service**.

```
Processing file deploy.wsdd
<Admin>Done processing</Admin>
```

Figure B.10: Output created by deployment command if successful.

Translator Web Service. Using these nine steps, you will be able to implement, deploy and run any Web Service you are planning to create in the future!

```
java -cp ".;axis.jar;commons-discovery.jar;
                commons-logging.jar;jaxrpc.jar;
                log4j-1.2.4.jar;saaj.jar;wsdl4j.jar;"
        ch.ethz.globis.demo.translator.TranslatorTextClient
        "Hello World!"
        "en"
        "de"
```

Figure B.11: Invocation of web service client.

```
Translation: Hallo Welt!
```

Figure B.12: Output of web service client invocation.

C Alternative Request Pattern

In this section we propose another request pattern that is based on `OMSJava` constructs such as `OMCollection` and `OMAssociation`. The query classes implementing this request pattern can easily be used instead of the query classes proposed in Sect. 3.2.
We believe that the request pattern proposed in this section can be used to allow remote querying on an OMS [13] database implementing an arbitrary schema. As explained in Chapt. 2 we assume a request factory that offers static methods customised to a particular schema and returning appropriate query objects. With the employment of our interaction architecture the factory is the only class bound to a database schema. Hence it becomes very simple to adjust a web or peer service to various database schemas. With this request pattern, queries of arbitrary complexity can be processed and their result returned within one request response cycle. This is achieved by allowing the nesting of query objects.

C.1 Introductory Examples

We derive our request pattern by defining some useful queries specific to the iServer schema. A complete presentation of the iServer schema can be found in Signer's publication [14]. Throughout this section we focus on the entity modeling and neglect the user management and layers schema.
An entity object can be identified by its name, its type and its sources / targets of all incoming / outgoing associations. Thus, if we are to design an entity query facility, we should offer the possibility to identify an entity through any of its attributes and any association in which entities are either a source or target collection. The signature of a method implementing entities selection could be designed as follows:

```
OMCollection selectEntities(  String name
                            ,  String type
                            ,  OMAssociation HasProperty
                            ,  OMAssociation HasSource
```

```
                              ,  OMAssociation HasTarget  )
```

This method would return an `OMCollection` containing all entity objects selected according to the parameter objects. For convenient usage we say that a null valued parameter does not affect the selection. The following example selects all entity objects named "African Savannah":

```
selectEntities( "African Savannah", null, null, null, null )
```

Proceeding in the same manner as for entity selection we define a query method specific to link objects. A link object can be identified through the properties inherited from the entity type and through members of the `HasSource` and `HasTarget` associations in which it must be in their domain. Following is the signature of the method:

```
OMCollection selectLinks(  OMCollection Entities
                         ,  OMAssociation HasSource
                         ,  OMAssociation HasTarget  )
```

The first parameter allows the selection of link objects according to their values of properties inherited from the entity type. We use the `selectEntities(...)` method defined above to select all links according to the entity attribute values. The resulting collection is given as the first parameter of the `selectLinks(...)` method. The evaluation of the latter includes a simple intersection of the `OMCollection` containing all entities selected by their attribute values and the Links collection containing all link objects. As an example we give the query for all link objects with an arbitrary name we refer to as [linkname]:

```
selectLinks(
   selectEntities( [linkname], null, null, null, null )
, null, null )
```

Now we have to define the evaluation of selection according to membership in domain or range collections of `OMAssociation` parameters. We derive the evaluation procedure using an example selection of `HasSource` associations. An association can be identified by a member of the domain and range collection. Hence, we define the signature of the method implementing the `HasSource` association selection as follows:

```
OMAssociation selectHasSource(  OMCollection domain
                              ,  OMCollection range  )
```

Since the domain and range collection of the `HasSource` association contains entity objects we can use the `selectEntities(...)` method to define the parameters. As an example we select all `HasSource` associations that point to the entity named "African Savannah":

```
selectHasSource( null,
   selectEntities( "African Savannah", null, null,
                         null, null )
)
```

The evaluation of this query can be accomplished by performing a range restriction on the `OMAssociation` collection requiring the range collections of the resulting associations to contain the same member entities as the collection resulting from the nested entity selection. In general an association selection is processed by performing a domain restriction with the domain parameter and a range restriction using the range parameter.
So far we have presented methods to select members of a collection according to attribute values, membership in another collection and a method to perform association selection. We give an example of how these methods can be combined to select all sources of links having entities named "African Savannah" as targets:

```
selectEntitities( null, null, null,
   selectHasSource(
      selectLinks( null, null,
         selectHasTarget( null,
            selectEntities( "African Savannah", null, null,
                                 null, null )
         )
      )
   , null )
, null )
```

The innermost query returns a collection containing all entities named "African Savannah" which is used as the range restriction criteria for the `selectHasTarget(...)` method. The collection of associations returned by the latter is intersected with all `HasTarget` associations in the `selectLinks(...)` method which returns a collection containing the link objects resulting from the intersection. This collection is now used to perform a domain restriction on the `HasSource` associations. The resulting collection of associations is intersected with all `HasSource` associations in the outermost query which returns the resulting entity objects as an `OMCollection`.

C.2 Generalised Query Model

After examination of the iServer API we observe that we can implement any querying API method as a combination of the three different kinds of query methods introduced in the previous section. These methods are: query by attribute value, query by collection intersection and query by association source / target collection intersection. In this section we propose Java classes implementing these queries which can be used to construct general queries independent from a particular schema. Instead of having a parameterised method processing the query we introduce uniform query objects initialised with the parameters. Thus, query processing can be initiated by invoking a method requiring no parameters.
Figure C.1 shows the UML diagrams of the Java classes required for the implementation. For now we disregard the fourth kind of query implemented by the `QueryByMatching` class. We also ignore the `QueryParameter` superclass common to all query classes in the beginning since its sense will become clear by explaining the functioning of the `QueryByCollection` class.
The simplest class is the one implementing the query by attribute. It defines members containing the name and value of the attribute after which objects of a particular type are to be

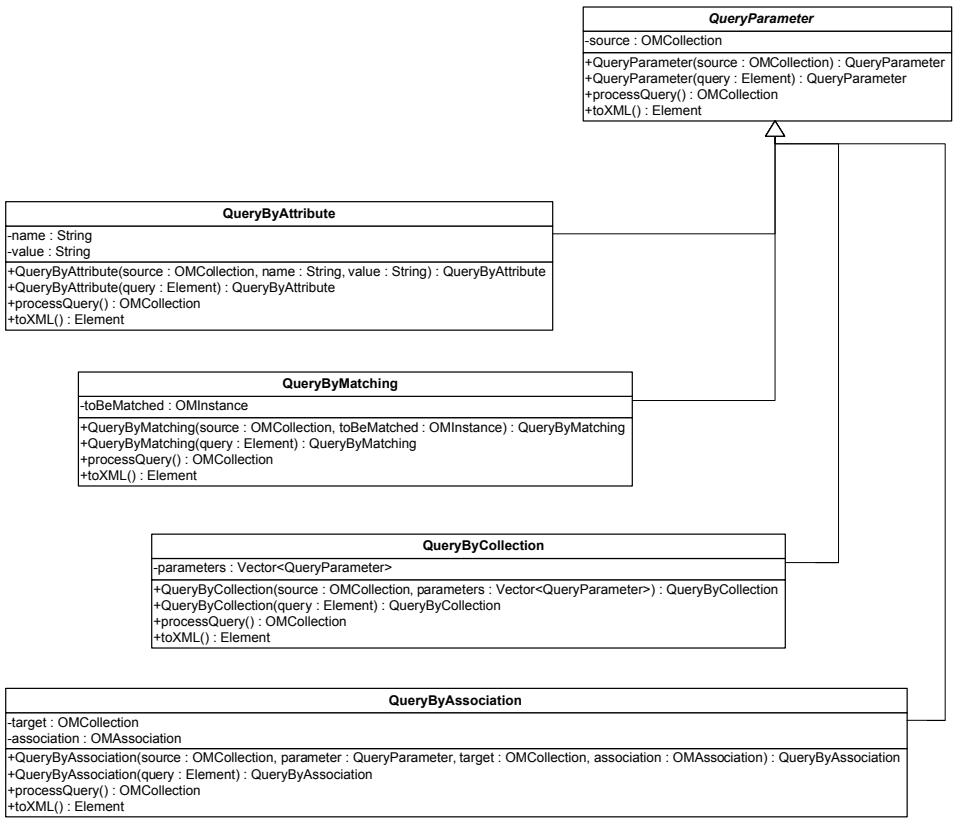

Figure C.1: UML definition of nested querying classes.

queried. Note that these members allow a `QueryByAttribute` object to be used to query for any type of object. Additionally, a member pointing to an `OMCollection` is inherited from the superclass `QueryParameter`. The `processQuery()` method returns an `OMCollection` containing all members of the member `OMCollection` `source` having the correct attribute value. We can express this query in AQL as follows (we make use of the names of the class member as defined in the UML diagram):

```
all O in source having( O.name = value )
```

Next we present the `QueryByCollection` class which implements the query by collection intersection. Its member property `parameters` is a vector of `QueryParameter` objects. The `QueryParameter` class is an abstract class from which all query classes inherit. Its main purpose within our current scope is to guarantee that every query class implements a `processQuery()` method returning an `OMCollection`. The query is evaluated by intersecting all parameter collections with the `source` collection inherited from the `QueryParameter` superclass. In AQL the performed query can be described as

```
source intersect parameters[0] intersect ...
         ... intersect parameters[n]
```

The third query class is a subtype of the `QueryByCollection` class called `QueryByAssociation`. As the name suggests it implements the query by association domain / range collection intersection. Since we never need to access an association explicitly within iServer we modified the query evaluation such that it does not return a collection of associations but the domain or range collection of the selected associations depending whether a range or domain restriction is performed. Thus we can intersect the returned collection with a collection resulting from any other kind of query and do not need to handle intersections of unary and binary collections. Note that the `source` collection inherited from the `QueryParameter` class always refers to the collection type returned by the `processQuery()` method. The `target` collection member is the one that is intersected with the parameter collection allowing a selection. In a first step we determine which of the member collections `source` and `target` correspond to which of the association's domain and range. Depending on the outcome of this determination, the query is evaluated as follows:

```
// if (source == association.domain)
// && (target == association.range) :

   domain( association range_restriction (
      target intersect parameter
   )

// else :

   range( association domain_restriction (
      target intersect parameter
   )
```

The last query class has been introduced because, in an iServer web or peer service, we need to match a local object with a remote object with which the remote iServer can be queried. For example, if we want to get all sources of remote links containing a local entity as target we first need to match the local entity to a remote entity before we can proceed querying for its sources. The `QueryByMatching` class implements such a matching facility. Its member `xmlRepresentation` is a representation of the local object. The `source` collection member inherited from the `QueryParameter` class is searched for an object matching the local object. The query evaluation returns an `OMCollection` containing the matching object(s).
Figure C.2 shows the XML schema definition of the XML elements returned by the `toXML()` method implemented by all query classes. The root element contains one top level query that can be of any query type. The result of this top level query is returned with the response message. The XML schema definition reflects the content container pattern implemented by the Java classes hierarchy shown in Fig. C.1. The attributes of the XML elements are not visible in the figure but they can be deduced from the Java classes. In the case of an application to the iServer the `xmlRepresentation` element corresponds to a JDOM element as generated by the iServer `OMInstance` objects.

C.3 Example Query

In this section we give an example query in terms of its XML representation (Fig. C.3)and the corresponding query object constructions. As opposed to the previous example we now query for all resources that are targets of links whose source entity has a name value ”African Savannah”.
A query factory method `selectTargetRessourcesBySourceEntity(String)` would be implemented as follows:

```
public static QueryParameter
   selectTargetRessourcesBySourceEntity(String entityName){

   QueryByAttribute selectByName =
      new QueryByAttribute("entities", "name",
                           "African Savannah");
   QueryByAssociation selectLinks =
      new QueryByAssociation("links", "entities",
                             selectByName, "hasSource");
   QueryByAssociation selectEntities =
      new QueryByAssociation("entities", "links",
                             selectLinks, "hasTarget");
   QueryByCollection selectRessources =
      new QueryByCollection("resources",
            new QueryParameter[] { selectEntities } );

   return selctRessources;
}
```

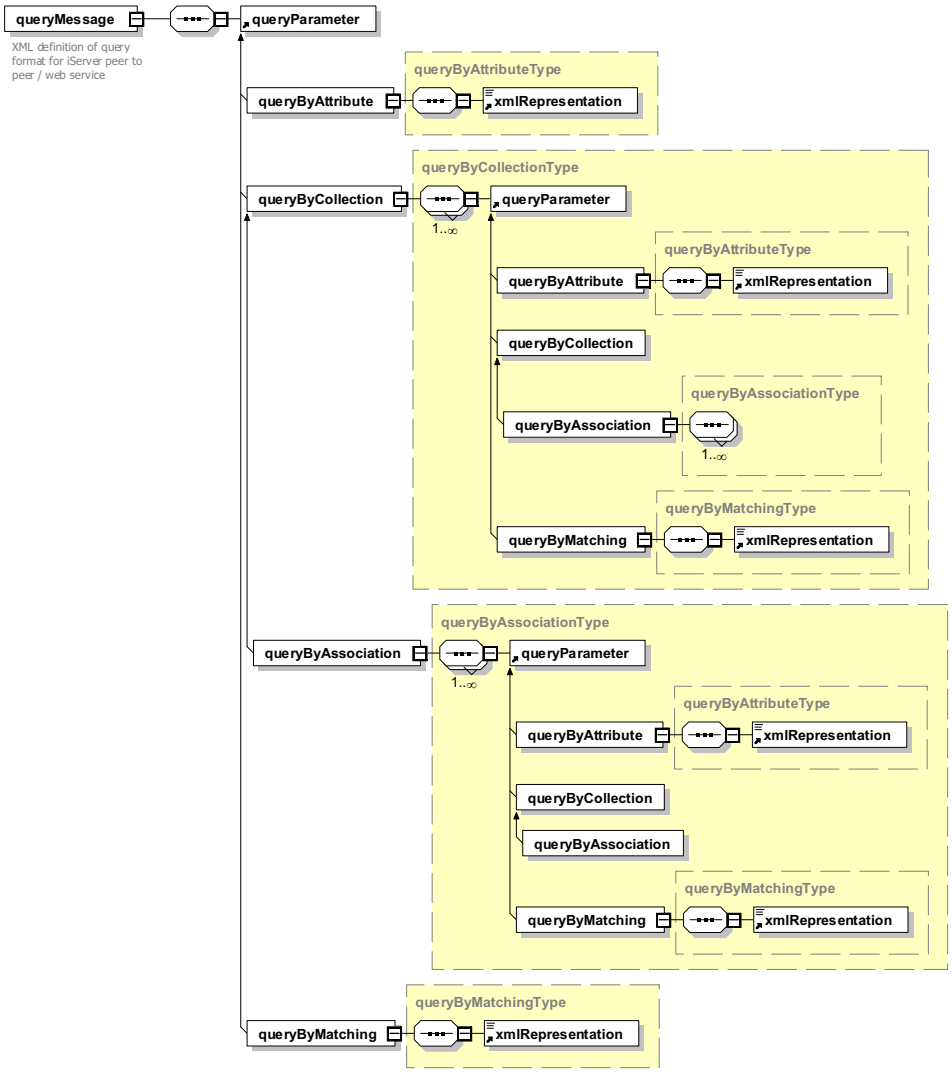

Figure C.2: XML schema definition for the XML elements generated by the `toXML()` method implemented by all query classes.

```
- <queryMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="query.xsd">
  - <queryByCollection sourceCollection="Resources">
    - <queryByAssociation sourceCollection="Entities" targetCollection="Links"
        association="HasTarget">
      - <queryByAssociation sourceCollection="Links" targetCollection="Entities"
          association="HasSource">
        - <queryByAttribute sourceCollection="Entities" name="name">
            <xmlRepresentation>African Savannah</xmlRepresentation>
          </queryByAttribute>
        </queryByAssociation>
      </queryByAssociation>
    </queryByCollection>
  </queryMessage>
```

Figure C.3: XML message encoding an example query according to the XML schema definition of the protocol schema

Acknowledgements

I would like to thank Beat Signer for supervising my diploma thesis. He patiently let me discover my joy for the area by establishing a perfect balance of freedom and duty. The meetings we had were always very helpful and I appreciated his feedback from reviewing my report.
I am grateful to Prof. Moira C. Norrie for repeatedly giving me the opportunity to work on a project and channeling my passions into accomplishing this thesis project. Furthermore, I want to thank Michael Grossniklaus for patiently answering all my questions, Rudi Belotti for inspiring me to design parts of my work as a general architecture and Melanie Raemy for suggesting a graph algorithm to resolve transitive user rating.
Thanks to Teodora Zamfirescu for correcting my report and sticking with me despite my self imposed unavailability in favour of this thesis. Finally, I would like to thank H. Dubach for his kind support during my studies.

Bibliography

[1] Apache Software Foundation: Web Services - Axis, Version 1.1 (http://ws.apache.org/axis/).

[2] Apache Software Foundation: Jakarta - Tomcat, Version 5.0.x (http://jakarta.apache.org/tomcat/).

[3] Brookshier, D. et al. *JXTA: Java P2P Programming*, SAMS, United Stats of America, 2002.

[4] Edmonds, J., Karp, R. *Theoretical improvements in the algorithmic efficiency for network flow problems*, Journal of the ACM, 1972.

[5] Flenner, R. et al. *Java P2P Unleashed*, SAMS, United States of America, 2003.

[6] Heinzer, C. *iServerP2P. Distributed iServer Architecture Based on Peer-to-Peer Concepts*, Inst. for Information Systems, ETH Zurich, 2004.

[7] JDOM Project (http://www.jdom.org).

[8] Sun Microsystems: Java Remote Method Invocation (http://java.sun.com/products/jdk/rmi/).

[9] Project JXTA, Version 2.3.3 (http://www.jxta.org).

[10] JUNG - Java Universal Network/Graph Framework, Version 1.6.0 (http://jung.sourceforge.net).

[11] Sun Microsystems *JXTA v2.3.x: Java Programmer's Guide* (http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf), 2005.

[12] Norrie, M.C. *Lecture Notes for ISG course, Chapt. 2: Data Modelling*, Inst. for Information Systems, ETH Zurich, 2001.

[13] Norrie, M.C. et al. *OMS Pro 2.0 Introductory Tutorial*, Technical Report, OMS Pro Version 2.0, ETH Zurich, 2003.

[14] Signer, B., Norrie, M.C. *A Framework For Cross-Media Information Management*, Proceedings of EuroIMSA 2005, International Conference on Internet and Multimedia Systems and Applications, Grindelwald, Switzerland, February 2005.

[15] Sun Microsystems: Sun Developer Network (http://forum.java.sun.com).

[16] Wilson, B. *Inside JXTA. Programming P2P Using The JXTA Platform* (http://www.brendonwilson.com/projects/jxta/), New Riders Publishing, 2002.