Ausarbeitung

CORBA Telephone-Company

APR - Angewandte Programmierung

HÖBERT Timon | SOCHOVSKY Josef
MONGIA Himanshu | PAWLOWSKY Gabriel

5BHITS

VERSION 1.0

Last Update: 10. April 2013

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 General Remarks

We suggest to read the following tutorials before you start implementing:

- Study the JacORB Programming Guide. Specifically chapters 4 and 5 are interesting for this lab. You can find it in the JacORB distribution in doc/ProgrammingGuide.pdf.
- The JacORBbank demo shows how to work with implicit transactions and is a good reference for this lab. You can find it in the JacORB distribution in demo/bank/transaction/implicit.
- IDL overview: Provides a good overview over the Interface Definition Language.
- Java IDL Mapping: Describes the mapping from IDL to Java.
- Java CORBA Introduction: This is a nice introduction for using CORBA with Java. Note that it's not JacORB specific!

Be sure to check the Tricky Parts section for questions!

1.2 Submission Guide

Submission

- Upload your solution as a ZIP file. Please submit only the sources including your build and idl file (see below) of your solution (not the compiled class files and no third-party libraries).
- Your submission must compile and run on our laptops. Use and complete the provided ant template.

Interviews

- After the submission deadline, there will be a mandatory interview.
- During the interview, you will be asked about the solution that you uploaded. In the interview you need to explain your code, design and architecture in detail.
- Be prepared that we will held reviews every lesson to get some additional marks.

1.3 Description

In this assignment you will learn:

- the basics of CORBA (Common Object Request Broker Architecture)
- how to write a CORBA solution by starting with an IDL (Interface Description Language)
- how to generate Java artifacts from CORBA
- how to implement simple CORBA objects and clients
- how to use implicit transactions in CORBA

1.3.1 Overview

CORBA objects are hosted by server applications that provide one or more object instances to clients. The communication of both the client and the server is done by using so called ORBs (Object Request Brokers) that deal with marshaling (encoding arguments in a platform independent-format) and unmarshaling (decoding platform-independent data type representations into platform specific formats) and invocation of CORBA operations. One primary advantage of CORBA compared to other technologies such as RMI, .NET Remoting, or Web services is its coverage of services. That is, CORBA supports the full set of additional services that are required in most distributed system environments such as naming and trading, transaction and synchronization, notification and security services.

In this assignment we will use CORBA to implement a short messaging service (sms) infrastructure for telephone companies (telcos). Since it must be possible for customers of one telco to send messages to customers of other telcos, these telcos must be able to communicate with each other. Each telco can have its own IT infrastructure, possibly implemented in different languages on different platforms. Therefore, a standard communication technology has to be used for integrating these heterogeneous solutions. We choose CORBA to realize such a telco system.

1.3.2 Domain assumptions

- A telco is a telephone company which provides its customers a service to send short messages. Each telco has a name and its own unique prefix which consists of several digits (but at least one) (e.g. "0664").
- Messages contain text of arbitrary length (quite contradictory to sms, but not relevant for our purpose) and get a timestamp on sending. Messages may be sent to multiple recipients, who may be customers of different telcos.
- A telco internal number is a telephone number that is only unique within a telco's scope and consists of several digits (but at least one) (e.g. "123456"). A full number is a telephone number that is unique accross telcos, i.e., it consists of a prefix and a telco internal number (e.g. "0664/123456").
- Customer authentication is based on the telco internal number together with a four digit PIN code.

1.3.3 Architectural considerations

In this assignment we won't have such a simple client-server-architecture as before. Instead there might be several different telcos, that have to communicate with other telcos over a well defined, public interface and with their clients over a telco specific interface.

So each telco has to serve two purposes: on the one side it has to deal with messages sent by its own customers and potentially has to forward it to other telcos, on the other side it must be able to handle messages coming from other telcos and forward them to its own customers.

So customers may login, logout and send messages using the internal interface of their telco. Ïnternalïn this context means "telco internal", therefore this communication could be handled using any possible protocol or implemented in any programming language (we will also use CORBA and Java for this). It first gets exciting when the telco server has to communicate with other telcos (for example if a recipient of a message is not a customer of the sender's telco) over a well defined public interface, that each telco has to support. To make it even more interesting, the delivery of the message to all recipients must be handled within an (implicit) CORBA transaction, i.e., either all recipients receive the message, or none. To sum it up, each telco has an internal interface for serving its own customer requests, and an external interface for forwarding incoming messages to its own customers.

So when a message is received on a telco, it has to deliver the message to the recipients. If the concerned recipient is currently online, the telco has to immediately forward it (therefore we need client callbacks again), otherwise it has to store it temporarily and transmit it the next time the recipient goes online. However, it's not necessary to store the state persistently when the telco is shut down.

As already said in the last assignment, most distributed object frameworks provide a naming service, which allows binding/looking up remote references to/by simple names. CORBA provides one (org.omg.CosNaming.NamingContextExt), and we will use it for binding telco references to their prefixes, allowing customers to find their telco and telcos to find other telcos. As you will see this happens quite similar as with the java.rmi.registry.Registry in RMI.

The following figure shows an example scenario:

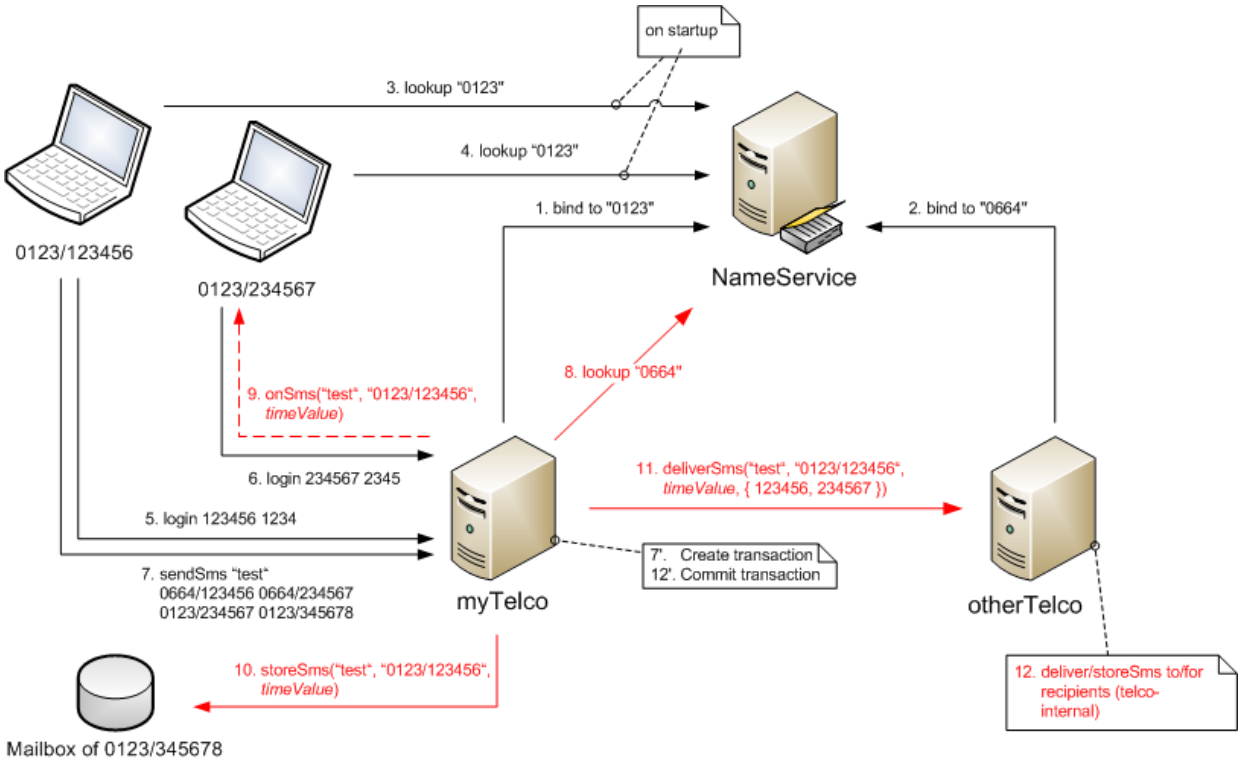

Abbildung 1: Telco Architecture

All operations displayed in red have to be executed within one single transaction (started in 7'), the one shown using a dashed line is a notification (thus called on the client callback object). The mailbox itself is not a stand-alone communication partner (such as client or server) - it's meant as a server-side data structure for storing missed messages of a single customer (anyway, you may implement this requirement however you want). Please note that the order of some operations (e.g. 9/10/11) is interchangeable and solely depends on your implementation decisions.

1.3.4 Server

In this assignment the server is simply a specific implementation of a telco. It allows customers to login, send sms and logout, as well as other telcos to transmit messages to this telco's recipients.

Arguments

Your server program shall expect exactly one argument (if missing, print a usage message and exit):

- fileName: specifies the name for a properties file that can be found on the classpath. The properties file then should be read in from the classpath (see the hint section for details). The format of properties file looks like the following:

```
#prefix for the telco, also used for binding it to name service
prefix:0664
#name of the telco
name:B2
#customer data follows (telco internal number:pin pairs):
123456:1234
234567:2345
345678:3456
...
```

Listing 1: Example Properties

An example file is provided and can be downloaded here. It contains the prefix (which shall be used for binding the telco object) and name of the telco to be started, as well as all customer data in form of telco internal number and pin code pairs. You can expect that there's no other text in it, so simply filter out the prefix and name and then read all accounts.

Interfaces

In CORBA you always have to define your interfaces in IDL files, thus, create the file telco.idl in your source folder (the ant template provided expects this path and name) and specify your interfaces, data structures etc. there. This specification is platform neutral and can be translated by IDL compilers to platform specific code (in our case: Java).

Public telco interface

This is the interface every telco has to implement to assure compatibility with other telcos. Normally, this interface would be given to you by some standardization group, but for our lab you also have to define it.

Required functionality

- Get the name of the telco.
- Get the prefix of the telco.
- Deliver message within a transaction: Allows other telcos to submit messages (consisting of text, the sender's full number and sending timestamp) to one or several recipients of this telco (in form of telco internal numbers). The telco has to forward the message to all online recipients and store it temporarily for offline recipients. However, both ways must be implemented transaction aware (see this for help)! In case a recipient is unknown or any internal error occurs the telco may raise an exception.

Internal telco interface

Extend the public interface to implement your own telco service. For any operation but login the client must be already logged in - you can check this either by using a session identifier or by letting the client submit its user name and password for each operation).

Required functionality

- Login of a customer: The client/customer has to supply its telco internal number, pin code and a callback object (see client description). The server has to check whether the submitted values are valid and return all missed messages.
- Logout of a customer: From now on the server has to store any incoming message for this customer until the next time he/she goes online.
- Send message as a single transaction: Lets the client send a message. The server has to create a transaction, possibly deliver the message to other telcos, possibly store the message for its own offline recipients and deliver it to its own online recipients. In case of any error (such as unknown number, unknown telco prefix or any failure, also of other telcos) the transaction must be rolled back, otherwise committed. You shall use the implicit transaction propagation model as described here. Note that also the sender may be a recipient.

Implementation details

On startup you first have to read in the properties file. Then create your telco object, make it remotely available by exporting it and bind it to the naming service using a NameComponent containing the prefix as id and "telco" as kind.

When a customer wants to send a message, your telco implementation has to create a transaction (see the JacORB bank example). Then it has to map recipients to telcos (using the full number's prefix) and look up the telcos in the name service (again by supplying the prefix). You shall cache already looked up telco references to avoid unnecessary lookups. If any problem occurs (such as telco could not be found in name service, telco does not know recipient number), you have to rollback the transaction, otherwise commit it. Note that also already looked up telcos might have gone offline - if a org.omg.CORBA.TRANSIENT or org.omg.CORBA.COMM_FAILURE exception occurs you may assume the telco is not reachable and have to rollback the transaction.

When the telco itself is part of a transaction (either when called by another telco or when a recipient of a message sent by an own customer is also an own customer), you first have to check whether such a number is known. If not, throw an appropriate exception (which will cause the transaction to rollback). If the recipient is currently online you have to transmit the message directly over the client callback (see below), otherwise store the message. Clients may also go offline without informing the server appropriately (for example if the computer crashes or the connection is lost). You don't have to detect this at once, but the next time the server is trying to contact the client to deliver a message: if a org.omg.CORBA.TRANSIENT or org.omg.CORBA.COMM_FAILURE exception occurs you may assume the client is offline. Therefore update the client's state and store the message to deliver it the next time the client goes online again.

As in the last assignment the telco has to manage state across several threads (they are managed by JacORB transparently). This again makes synchronization necessary when accessing your shared data structures!

If your server is ready for handling requests print "Server <prefix> up. Hit enter to exit." to the console and implement this behavior. On exit unbind the telco corba object from the naming service and also call ORB.shutdown(), otherwise your program might not shut down – again you must not use System.exit().

1.3.5 Client

In this part you have to build an interactive command line client application for your telco implementation.

Arguments

Again only one argument has to be specified (if missing, print a usage message and exit):

- prefix: the prefix of the telco to contact. The supplied value will be used for resolving the telco object in the naming service.

Callback interface

As you've already noticed, a telco has to inform its online recipients immediately on incoming messages. If you've solved the last assignment, you are already familiar with the callback concept - in CORBA it basically works exactly the same way as in RMI. To make it short: you have to define an interface in your IDL file (telco.idl) which your client has to implement.

Required functionality

- Receive message within a transaction: This method can be called by the telco to inform the client about a message. The client has to output the message to the console, containing the sending time, the sender's full number and the text (e.g. "20.10.2008 @ 12:01 | 0664/123456: This is a test message."). Please note that the client notification must also be part of the CORBA transaction originally issued by the sender's telco.

Implementation details

On startup of your program you first have to look up your telco in the name service using the specified prefix. Note that you can only cast the looked up org.omg.CORBA.Object instance by using the *Helper.narrow() method. If the telco was looked up successfully print "Contacted telco «telcoName>"(<telcoPrefix>) successfully."to the console. Afterwards export your client callback object like you exported your telco server object. The only difference is that you should not bind it to the name service, but instead simply pass it as parameter for the login command.

Interactive commands

The following commands must be supported by your client application. Take care of handling invalid commands and arguments and provide usage messages in these cases. Print meaningful error messages whenever the server throws an "expected" exception (such as "Invalid

login data." when trying to login with an invalid number/pin combination). Also print success messages if an operation was executed successfully. Note that you first have to login before sending sms (and, obviously, before logging out).

- **login <telcoInternalNumber> <pin>**
  Logs the user in. The telco server has to return all missed messages. The output should include the same information as displayed below:

  :> login 123456 1234
  Logged in successfully.
  Missed messages:
  20.10.2008 @ 12:01 | 0699/123456: This is a test message.
  20.10.2008 @ 12:05 | 0699/123456: This is also an extremely exciting test message.

- **logout**
  Logs out the currently logged in user.

- **sendSms «text>"[<telPrefix>/<telNumber>]+**
  Sends an sms containing the specified text (all text between (exclusive) the first two occuring quotes)) to a list (containing at least one) of space separated recipients (in form of full numbers in the specified format).

  :> sendSms "This is a test message."0664/123456 0664/234567
  Message sent successfully.

- **stop**
  Stops the client application. If a user is currently logged in you have to log it out implicitly. Again you may not use System.exit(), but have to orderly close all acquired resources (shut down the ORB).

1.3.6 Ant template

As in the last assignment we provide a template build file (build.xml) in which you only have to adjust some class names. Put your source into the subdirectory "src", place the *.properties files and telco.idl into the "src" directory and move on the command line to the directory where the build file is located.

Simply type "ant idl-compile" for idl compilation (compiles the telco.idl file) and "ant" for java compilation (this task also copies your .properties files to the build directory).

Type "ant run-server -Dprops=b2" to start a server reading from b2.properties. In the provided build file an argument is expected for the server (<arg value="${props}.properties"/>). By specifying the -Dprops=b2 option, ant actually passes "b2.propertiesäs an argument to your server.

Type "ant run-client -Dprefix=0664" to start a client that connects to a telco bound to 0664 (by substituting <arg value="${prefix}"/> in the build file by "0664").

Put the src directory including telco.idl and build.xml into your submission. Note that it's absolutely required that we are able to start your programs with these predefined commands!

1.3.7 Transactions help

How to provide transaction support in your telco/client

As the deliver message operation of the public telco interface and the message received operation of the client callback interface have to be executed within transactions, the following problem arises: According to the OMG Transaction Service Specfication a resource may only be involved in one transaction at the same time. However, we want our client and especially our telco to be able to receive messages from different transactions concurrently, otherwise the performance would be quite bad.

So don't let your public telco interface and client callback directly extend CosTransaction::Resource and CosTransaction::TransactionalObject, and instead create an additional interface that extends these both and only provides a simple method that takes a message parameter (containing sending time, sender's full number and text).

On the client's side it becomes now simple. Provide an implementation for the specified resource interface, which on commit simply outputs the received message. In your callback you then create a new instance of this for each incoming request, hand over the message, and register it with the coordinator.

On the telco's side it's a bit more complicated, since there might be several recipients that might be online or offline. If a recipient is online - there's no problem, just forward it to the callback object (which handles the transaction as described above). But if we have to store it for a recipient, we also have to consider the transaction semantics. Therefore also provide an implementation for the specified resource interface, which on commit stores the message to the recipient's mailbox. In your telco do the same as on the client's side (create resource, hand over message, register it with coordinator).

For your information: The CosTransaction::TransactionalObject is just a markup interface and hence has no operations at all (primary a historic relict from previous CORBA standards). However, you have to implement the 5 operations of the Resource interface. You can skip implementing the forget() operation and implement the commit_one_phase() operation as in the JacORB sample. The prepare() method shall return one of the three values Vote.VoteCommit, Vote.VoteRollback, Vote.VoteReadOnly depending if the changes shall be committed, or rolled back. Vote.VoteReadOnly shall be used if there were no changes at all.

Implicit transactions

Using implicit transactions means that transaction contexts storing the current state of a transaction are not directly visible in client or server code but is automatically added by so-called interceptors - in CORBA interceptors hook in requests and may modify the transmitted data such as arguments or may add service contexts such as in our case.

To inform an ORB that it shall use the already predefined interceptor for implicit transactions the following value for the property org.omg.PortableInterceptor. ORBInitializerClass.TSClientInit must be set during the ORB initialization: org.jacorb.transaction. TransactionInitializer. This is a class that registers an object of type org.omg.CosTransactions.Current as an initial reference named TransactionCurrent. This reference can then be retrieved with orb.resolve_initial_references. The provided ant template already sets this property (see the run-server and run-client task).

The begin operation of the org.omg.CosTransactions.Current interface may be used to initiate a transaction. The timeout shall be set to a value of 40. This has to be set before you begin a transaction. After a transaction has begun you can call whatever transaction aware functionality of CORBA objects you want. The transaction context is transmitted implicitly. For rolling back a transaction you have to call the rollback function (surprisingly). Committing the transaction is done via the commit operation (use true as argument). Never kill a server (regardless how) that has started a transaction before the timeout has passed. When you do manual tests you can set the timeout to a smaller value.

From transaction aware objects you have to use the org.omg.CosTransactions.Current interface, too. However, this time you shall receive the org.omg.CosTransactions.Control interface via the Current's get_control() operation. Control supports two different functionalities: returning a Coordinator (with get_coordinator()) and returning a Terminator (with get_terminator()). In transaction aware operations you have to register the transaction aware CORBA object with the coordinator's register_resource() operation. The Resource's operations (prepare, rollback, commit) are then used automatically when the transaction initiator calls commit or rollback. In case of any TransactionServer related exception (example: Inactive) throw the runtime exception org.omg.CORBA.TRANSACTION_ROLLEDBACK. To get the exact syntax of the transaction server related operations take a look at the CosTransactions.idl or study the OMG Transaction Service Specification.

1.3.8 IDL compilation

For compiling your IDL file, which uses transaction service interfaces, you have to

- write #include <CosTransactions.idl> at the top of your IDL file to include the interface definitions
- and therefore tell the JacORB IDL compiler where to find the required IDL file <CosTransactions.idl>: Use the -I option (e.g. -I/opt/jacorb/idl/omg/) to do this. If you use our ant template this option is already included (take a look at the idl-compile target).

1.4 Hints & Tricky Parts

- Study the JacORB programmer's guide, in particular chapter 4 - Getting Started - explains how to program a CORBA server and a CORBA client with JacORB. Chapter 5 explains how to access the name service. Chapter 4 contains everything you need to implement your server and client (except the transactional stuff).
- In order to use JacORB instead of the built-in (and incomplete) ORB of SUN that is provided with Java, you have to provide two arguments to the Java VM with -D:
  - -Dorg.omg.CORBA.ORBClass=org.jacorb.orb.ORB and
  - -Dorg.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton

- You also have to include all the JacORB JAR files (located in the lib directory of the JacORB installation) into the CLASSPATH. Instead of manually applying these settings, you can also use the jaco shell script that is located in the $JACORB_HOME/bin directory. This script invokes Java with the appropriate VM arguments and classpath settings. We suggest using the provided ant template, which already includes these settings.

- **JacORB services**: You will need to start the naming and transaction service to accomplish this task. All described service commands are located in the $JACORB_HOME/bin directory. We recommend using the predefined tasks of the provided ant template which clearly simplify the configuration and startup of these services.

  - **Naming service**: You need the naming service to bind and lookup your CORBA objects (this is quite similar to the RMI registry from the last lab).
    Usage: ns -Djacorb.naming.ior_filename=$HOME/NS_REF or ant run-ns
    If you don't use the provided ant target, you have to copy the jacorb_properties .template from $JACORB_HOME/etc to the directory where you execute your servers/clients and rename it to jacorb.properties. Then edit the ORBInitRef. NameService entry and set it to the appropriate path
    (e.g. ORBInitRef.NameService=file:///dslab/home/dslab/dslabXXX/NS_REF), so your programs are able to locate the naming service. There you can also edit a lot of other configuration parameters when invoking JacORB, although you don't have to.
  - **Transaction service**: You first have to start the naming service as described above.
    Usage: ts -ORBInitRefNameService=file://$HOME/NS_REF or ant run-ts
  - **Naming manager**: If you are working at home (this one's a GUI application) you can use this tool to view the objects bound to the naming service. This is helpful if you are unsure about your binding code (the transaction service is also visible there if it was started successfully).
    Usage: nmg -ORBInitRefNameService=file://$HOME/NS_REF or ant run-nmg

- **Using JacORB at home**:

  - Download JacORB. If you want to build the JacORB API documentation (see below) you have to select the source distribution, otherwise the binary one is sufficient.
  - Set the environment entry $JACORB_HOME.
  - If you don't want to use the predefined ant targets you have to call "ant jaco" in JACORB_HOME and should include $JACORB_HOME/bin into your $PATH variable.

- Reading in a properties file from the classpath (without exception handling):

```
java.io.InputStream is = ClassLoader.getSystemResourceAsStream("b2.properties");
if (is != null) {
    java.util.Properties props = new java.util.Properties();
    try {
        props.load(is);
        String prefix = props.getProperty("prefix");
        ...
    } finally {
        is.close();
    }
} else {
    System.err.println("Properties file not found!");
}
```

Listing 2: Reading Properties

- Submitting dates in CORBA: simply pass the time value as returned by java.util.Date.getTime() and reconstruct the date using the java.util.Date(long time) constructor. Note that a long in Java is equivalent to a long long in the IDL.

1.5 Further Reading Suggestions

APIs

- CORBA: Package API, ORB API, CORBA Object API
- CORBA Naming Service: Package API, NamingContextExt API
- CORBA Portable Server: Package API, POA API, Servant API
- JacORB and CORBA Transactions: you have to call "ant doc" in JACORB_HOME (works only with the source distribution). Afterwards the API can be found in JACORB _HOME/doc/api/ - mainly the org.omg.CosTransactions package will be of interest for you.
- Properties: Properties API

Specifiations

- OMG CORBA Specification: This is the original OMG CORBA Specification. You can use it as a reference if you are specially interested in CORBA.
- OMG Transaction Service Specfication: This is the original OMG Transaction Service Specification. In general, it is not necessary to read all of this. However, in Section 2 all involved CORBA interfaces are explained (as Resource, Control, Coordinator, Terminator).
- OMG IDL to Java Mapping Specification: The official OMG document about the mapping from IDL to Java.

Tutorials:

- CORBA Callbacks: Simple tutorial for callbacks (not JacORB specific).
- Java CORBA Guide: Here you can find several informations about CORBA with Java; again this is not related to using JacORB.

Proposed Literature:

- Advanced CORBA(R) Programming with C++ by by Michi Henning and Steve Vinoski

1.6 Submission Guide

Submission

- Every group must have its own design/solution! Meta-group solutions will end in massive loss in points!
- After the design review with the team leader, any design changes must be approved. Write a change request and describe what parts you want to change and why you think it's necessary to adopt your design decisions.
- The work packages must be provided with estimated times immediately after the design phase.
- As for group work usual, a protocol with the UML-Design, the work-sharing, the timetable and test cases is mandatory!
- Upload your solution as a ZIP file. Please submit only the sources of your solution and the build.xml file (not the compiled class files and no third-party libraries).
- Your submission must compile and run! Use and complete the provided ant template.
- Before the submission deadline, you can upload your solution as often as you like. Note that any existing submission will be replaced by uploading a new one.

Interviews

- After the submission deadline, there will be a mandatory interview.
- The interview will take place in the lesson. During the interview, every group member will be asked about the solution that everyone has uploaded. Changes after the deadline will not be taken into account! There will be only extrapoints for nice and stable solutions. In the interview you need to explain the code, design and architecture in detail.

Points

Following listing shows you, how many points you can achieve:

- **Documentation(15):** timetable(2), explanation(4), description(JavaDoc)(4), protocol requirements(2), tests(3)
- **Implementation(36):** client(8), conditions(8), transactions(8), collections(6), services(6)
- **Design(8):** uml-class(5), usecase(3), +extrapoints activity(+2)
- **Testing(16):** ant(2), arguments(2), unit-testing(6), login(1), logout(1), sendSMS(2), stop(1), error handling(1)

distributed by: Vienna University of Technology
Insitute of Information Systems 184/1
Distributed System Group [1]

2 Designüberlegung

Folgende Funktionalität soll bereitgestellt werden:

TelNr
-prefix:String
-suffix:String

Message
-text:String
-sender:TelNr
-time:long
-receiver:TelNr[]

IClient
+onSMS(m:Message):boolean

IPublicTelcoI
+getName():String
+getPrefix():String
+deliver(m:Message):boolean

IPrivateTelcoI
ArrayList<Message> login(nr:String,pin:String,callback:IClient):boolean
+logout(nr:String,pin:String)
+send(nr:String,pin:String)

Telco
-users: ConcurrentHashMap<TelNr,User>

User
-nr:String
-prefix:String
-pin:String
-callback:IClient
-mailbox:ArrayList<Message>

PropertiesReader

2.1 IDL-File

```
#include <CosTransactions.idl>

module telco {
    module general {
        module gen {
            struct TelNr {
                string prefix;
                string suffix;
            };

            typedef sequence<TelNr> TelNrArray;

            struct Message{
                string text;
                TelNr sender;
                long long time;
                TelNrArray reciever;
            };

            typedef sequence<Message> MessageArray;

            exception InvalidLoginException {
                string reason;
            };

            exception DeliverException{
                string reason;
            };

            interface IClient{
                boolean onSMS(in Message m);
                boolean onSMSExplicit(in Message m, in CosTransactions::Control control_);
            };

            interface IPublicTelco {
                string getName();
                string getPrefix();
                boolean deliver(in Message m) raises(DeliverException);
                boolean deliverExplicit(in Message m, in CosTransactions::Control control_)
                    raises(DeliverException);
            };

            interface IPrivateTelco : IPublicTelco { // IPrivateTelco extends IPublicTelco
                MessageArray login(in string suffix, in string pin, in IClient callback) raises(
                    InvalidLoginException);
                boolean logout(in string suffix, in string pin) raises(InvalidLoginException);
                boolean send(in string suffix, in string pin, in Message m) raises(
                    InvalidLoginException, DeliverException);
                boolean sendExplicit(in string suffix, in string pin, in Message m) raises(
                    InvalidLoginException, DeliverException);
            };

            interface ITransaction : CosTransactions::Resource, CosTransactions::
                TransactionalObject{
                void useMessage(in Message m);
                void useMessageExplicit(in Message m, in CosTransactions::Control control_ );
            };
        };
    };
};
```

Listing 3: IDL File

2.2 Client

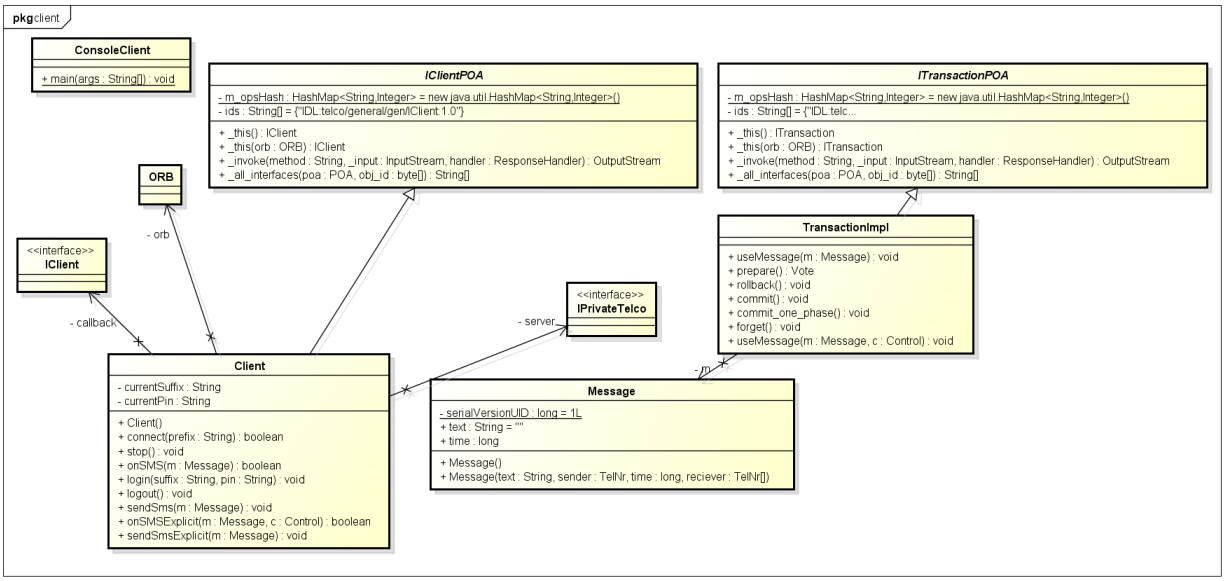

Abbildung 2: UML-Klassendiagramm Client-Paket

2.3 Server

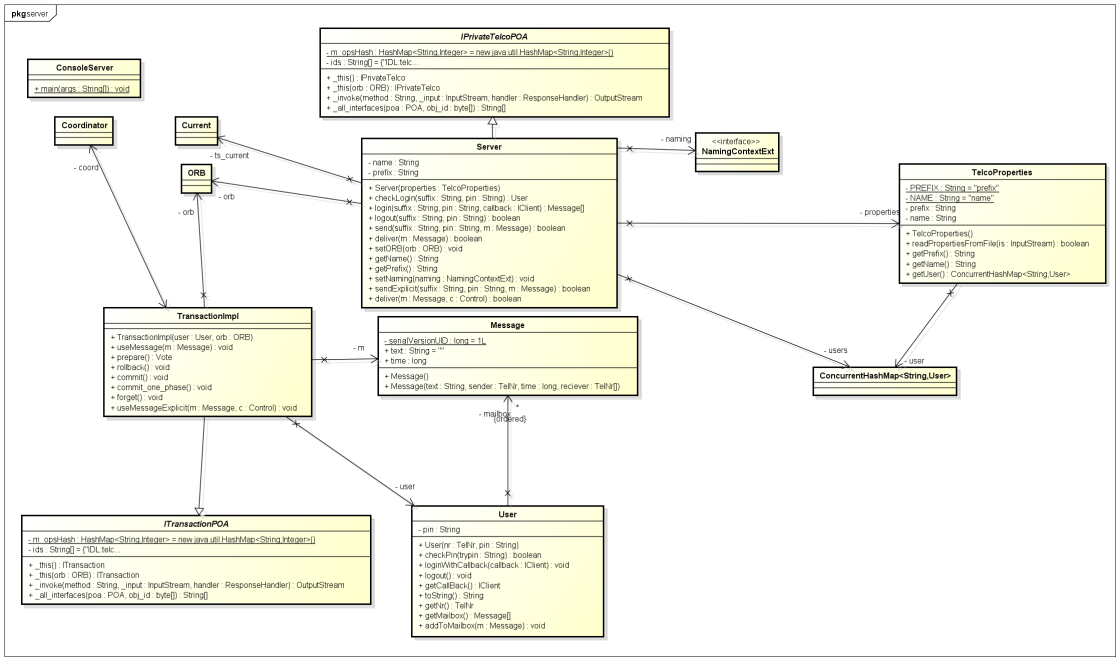

Abbildung 3: UML-Klassendiagramm Server-Paket

2.4 General

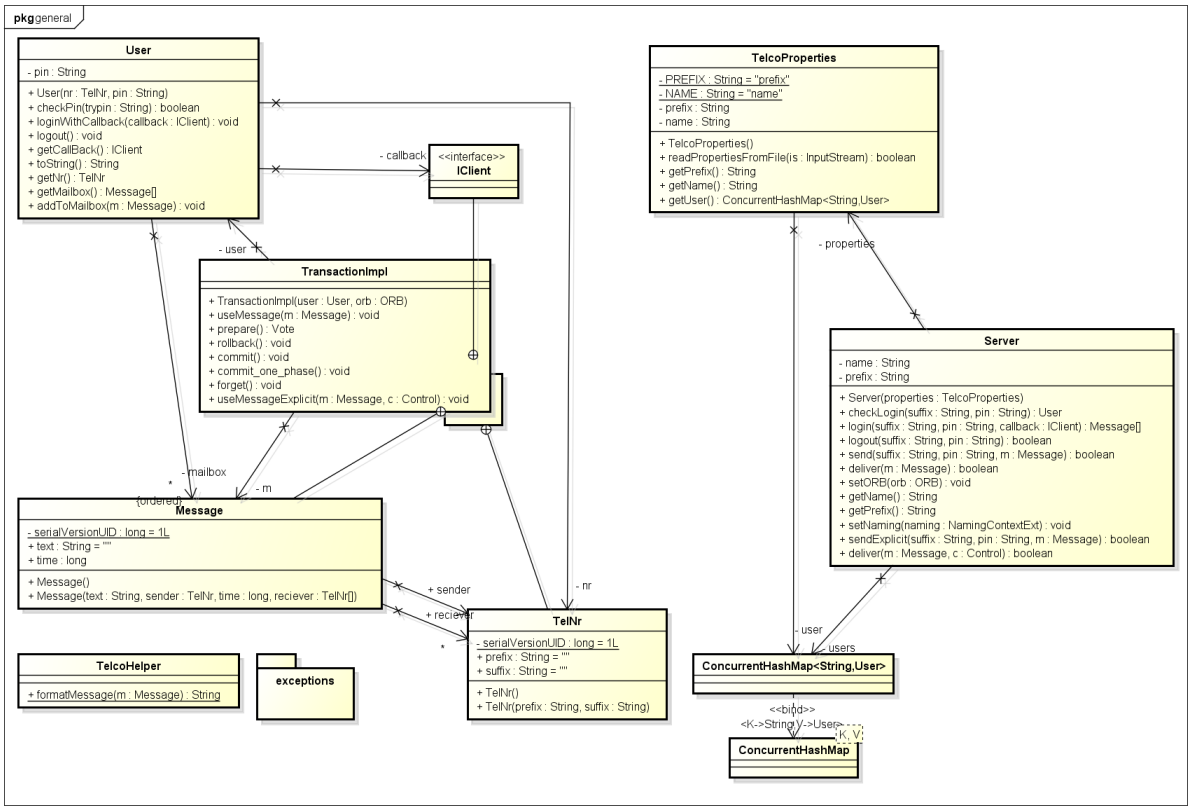

Abbildung 4: UML-Klassendiagramm General-Paket

2.5 Aktivitätsdiagramm

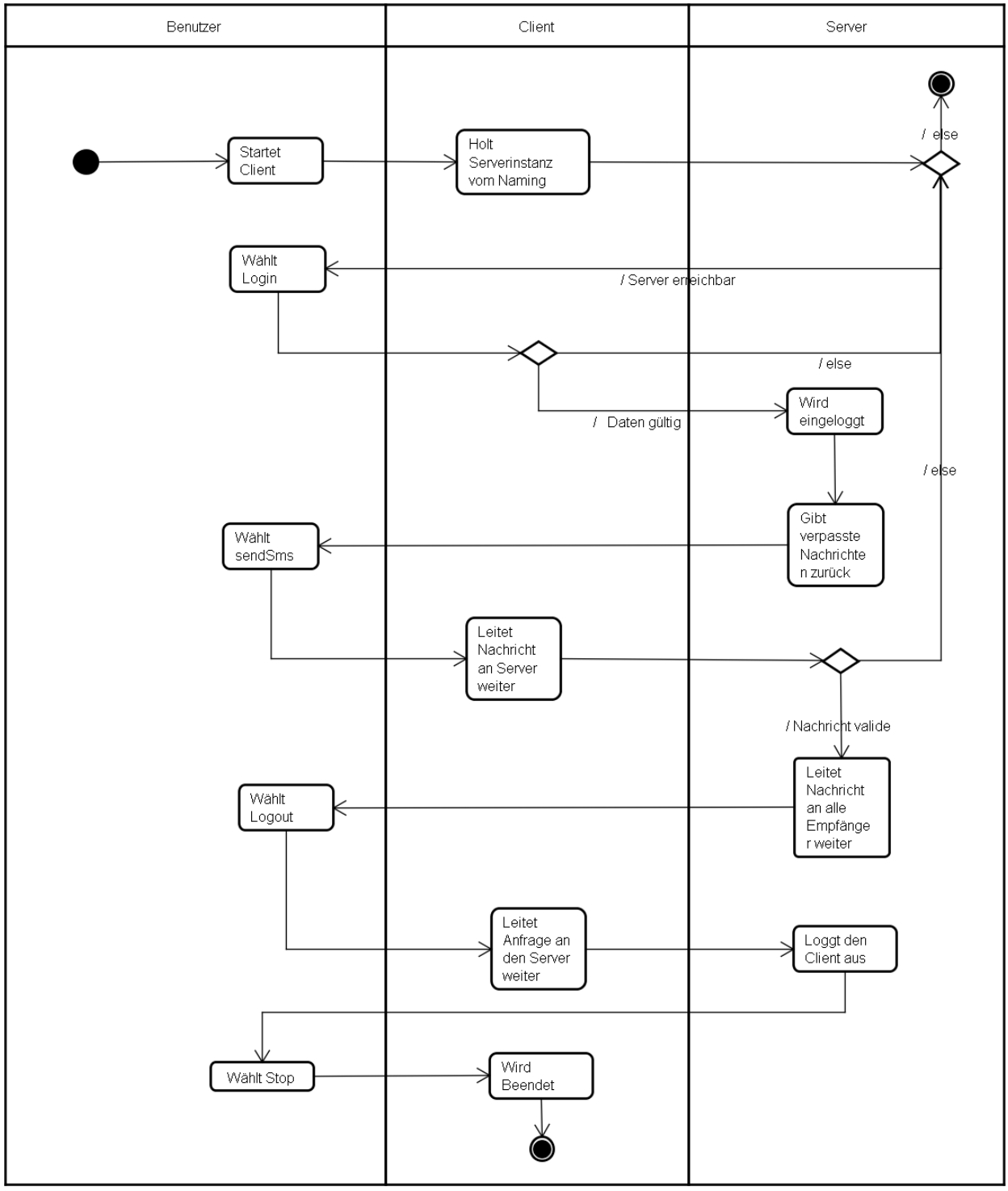

Abbildung 5: Aktivitätsdiagramm

2.6 Use-Case-Diagramm

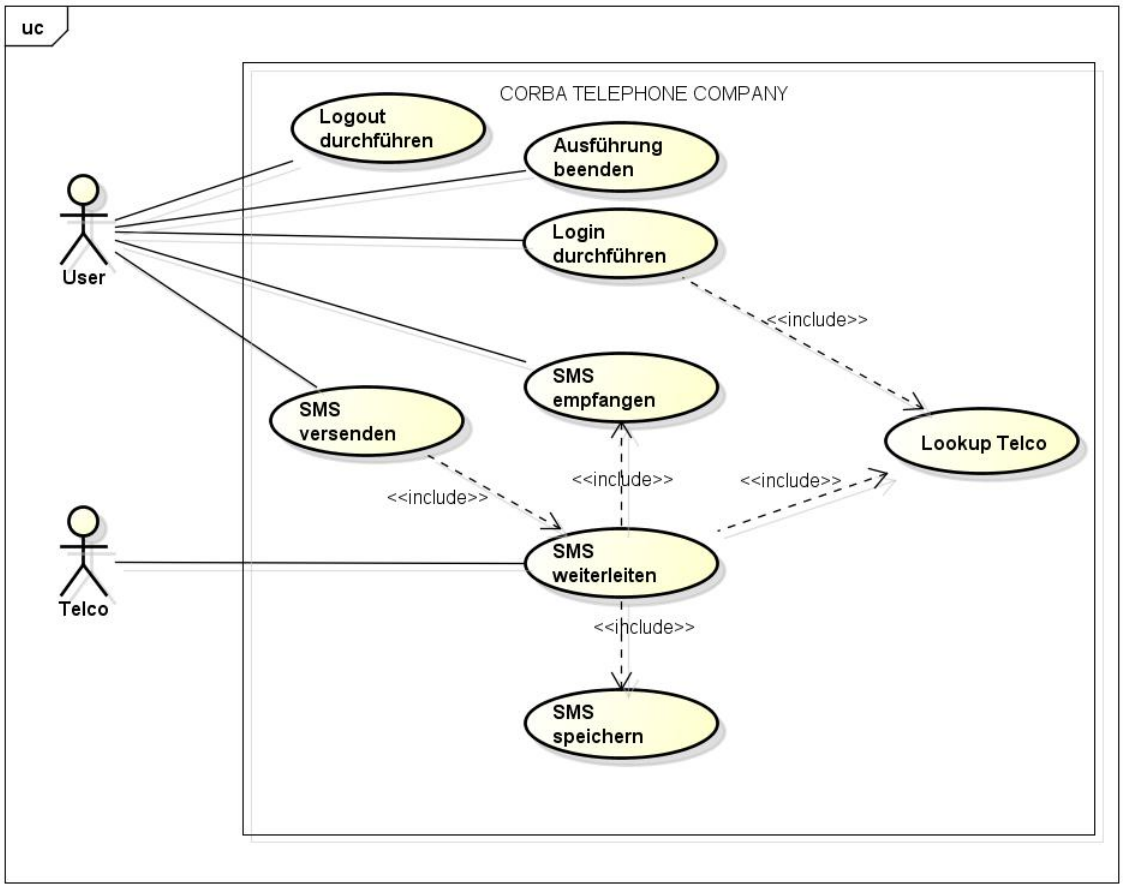

Abbildung 6: Use-Case-Diagramm

3 Installation JacORB

Voraussetzungen

Laufende Version von ANT

```
ant -version
Apache Ant(TM) version 1.9.0 compiled on date
```

Außerdem sollte man die momentane Version von den JacORB Sources bereits heruntergeladen haben (v3.2).
Eine weitere Voraussetzung ist die Installation des aktuellen JDK's (v1.7), sowie die Möglichkeit diese über die Konsole ansprechen zu können.

```
java -version
java version "1.7.0_01"
Java(TM) SE Runtime Environment (build 1.7.0_01-b08)
```

3.1 Installation

Erzeugen des jacorb.properties-Files

In dem entpackten JacORB-Ordner soll in dem Unterordner /etc ein "jacorb.propertiesFile erzeugt werden, dieses sollte folgenden Inhalt haben:

```
org.omg.CORBA.ORBClass=org.jacorb.orb.ORB
org.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton
```

Wechseln in den JacORB-Ordner

Nun muss man mit der Konsole in den JacORB-Ordner wechseln. Dies kann mit den folgenden Eingaben erreicht werden:

```
set JACORB_HOME=C:\JacORB
set PATH=\%PATH\%;\%JACORB_HOME\%\
cd JACORB_HOME
```

Kompilieren von JacORB

Nun wird JacOrb mit ANT kompiliert

```
ant
```

oder

```
ant -all
```

Testen von JacORB

Um das kompilierte JacORB nun zu testen muss der folgende Befehl die Synopsis zurückgeben: jaco

3.2 Problemlösungen

ant

Sollte ”ant -version” nicht funktionieren sollten folgende Eingaben unter Windows 7 in derselben Konsole durchgeführt werden:

```
set ANT_HOME=C:\
set PATH=\%PATH\%;\%ANT_HOME\%\
```

Kompilieren

Sollte das Kompilieren sofort abbrechen, sollte überprüft werden ob die JacORB Sources richtig heruntergeladen wurde und nicht JacORB bin-Files.

4 Arbeitsaufteilung & Endzeitaufteilung

Pro Aufgabenpunkt wird Inline-/Protokolldokumentation sowie Testing inkludiert.

Aufgabe	**Person**	**Soll**	**Ist**
Installation JacORB	Josef	3h	3h
Properties Reader	Timon	2h	2h
CORBA Grundimplementierung	Timon	5h	7h
Callback Implementierung	Timon	2h	2h
Ant Konfiguration	Timon	3h	4h
(Console-)Client	Gabriel	5h	6h
(Console-)Server	Herry	5h	6h
(Console-)Server	Josef	5h	4h
Transaktionen	Josef	3h	4h
Transaktionen	Herry	3h	5h
Explizite Transaktionen	Gabriel	3h	5h
SUMME Timon	Timon	12h	15h
SUMME Herry	Herry	8h	11h
SUMME Gabriel	Gabriel	8h	11h
SUMME Josef	Josef	11h	11h

Tabelle 1: Arbeitsaufteilung mit Aufwandsabschätzung & Endzeitaufteilung

5 Implementierung

5.1 Server

Der Server muss folgende Fuktionen bieten:

- Informationen weitergeben
  - Name
  - Preifix
- Nachricht von einem User annehmen
- Nachricht verarbeiten
  - An einen anderen Server weiterleiten (Server-Server Lookup)
  - An ein Client-Callback weiterleiten
  - Speichern der Nachrichten für Offline-User

5.1.1 Informationen weitergeben

Der Server muss seinen Namen (beispielsweise "BOB") und seine Prefix (beispielsweise "0664") an andere Server und Clients weitergeben koennen. Um diese Funktionalität zu unterstützen werden die Methoden getName() und getPrefix() der IDL-Datei im Server implementiert.

5.1.2 Nachrichten von einem User annehmen

Der User ruft am Server die Methode send() auf, übergibt dieser eine Nachricht die einen Inhalt und möglicherweise mehrere Adressanten hat. Diese Methode startet eine Transaktion. Wenn es zu komplikationen während der Bearbeitung der Nachricht kommt erfährt der User über den Fehlschlag des Sendens üder die DeliverException.

5.1.3 Nachrichten verarbeiten

Nach dem Empfangen der Nachricht hat der Server nun die Aufgabe die Nachricht an alle angegebenen Benutzer weiterzuleiten.

An einem entfernten Server weiterleiten
Ist der Adressant nicht mit der lokalen Prefix angegeben, muss der Server den verantwortlichen Server kontaktieren. Is es nicht möglich den entfernten Server zu erreichen wird das Verschicken abgebrochen. Um nicht bei jeder zu sendenden Nachricht zum entfernten Server neu lookupen zu müssen wird bei einem gefundenen Server eine Instanz dessen gespeichert um ihn bei einem zweiten Verbindungsversuch schneller ansprechen zu können.

An ein Client-Callback weiterleiten
Bei einem erfolgreichen Login wird ein Callback zu einem Client gespeichert, das bedeutet er ist über ein gespeichertes Objekt erreichbar. Wird eine Nachricht für den Client empfangen erhält er diese innerhalb derselben Transaktion. Ausgegeben wird die Nachricht dann bei einem erfolgreichen Commit serverseitig.

Speichern der Nachrichten für Offline-User
Wenn beim Server kein Callback für den zu erreichenden User gespeichert ist, wird eine Nachricht in seine Mailbox gespeichert. Bei einem erfolgreichen Login werden ihm dann im Anschluss alle zwischengespeicherten Nachrichten ausgegeben. Aufgrund der Aufgabenstellung war es nicht notwendig die Mailbox zu persistieren.

5.2 Client

Es gibt einen ConsoleClient. Dieser öffnet eine Konsole, in der die Kommandos stop, login, logout und sendsms eingetippt werden können. Dazu muss er sich zu Anfang eine ORB-Referenz vom Server holen. Dadurch ist er mit diesem verbunden. Danach müssen für die Kommandos folgende Funktionen zur Verfügung stehen:

- Login ermöglichen
  - Suffix
  - Pin
- Logout ermöglichen
- Nachricht senden
  - Nachrichteninhalt
  - Liste der Empfänger

Login ermöglichen
Wenn der User den Login-Befehl mit einem Suffix und einem PIN aufruft, wird zuerst auf den ORB zugegriffen, sofern die Daten valide sind. Dann wird der User am Server eingeloggt und kann Nachrichten versenden, oder sich ausloggen. Sind die Userdaten allerdings nicht valide, wird eine Exception geworfen. Beim Login wird außerdem ein Callback des Clients an den Server übergeben, sodass dieser auf den Client zugreifen kann.

Logout ermöglichen
Wenn der User den Logout-Befehl eintippt, wird zuerst überprüft ob er überhaupt eingeloggt ist und gegebenenfalls eine Exception geworfen. War er zuvor eingeloggt, wird am Server ausgeloggt und damit wird das Client-Callback gelöscht.

Nachricht senden
Dabei kann der User eine Nachricht mit doppelten Hochkommas und danach so viele Empfänger wie er möchte angeben. Diese werden dann intern in ein Message-Objekt geparst, dass

eine Liste aller gewünschter Empfänger sowie den Nachrichteninhalt speichert. Dieses wird, sofern der User eingeloggt ist, dann an den Server geschickt. Dieser leitet die Nachricht dann an alle gespeicherten Empfänger weiter.

5.3 Transaktionen

5.3.1 Allgemein

Um mit Transaktionen zu arbeiten muss die jeweilige Klasse das Interface **CosTransactions::Resource** und **CosTransactions::TransactionalObject** implementieren. Das Resource Interface legt Transaktionsmethoden wie commit, rollback, commit_one_phase, usw. fest, wogegen das TransactionalObject bloß ein Markup-Interface ist.

5.3.2 Implizite Transaktionen

Eine implizite Transaktion bedeutet dass die Übergabe des Transaktionskontexts nicht direkt im Code ersichtlich ist, sondern von sogenannten **"Interceptors"** automatisch hinzugefügt wird. Um einem Object Request Broker mitzuteilen, dass die bereits vordefinierten Interceptors für implizite Transaktionen verwendet werden sollen, muss die Eigenschaft **org.omg.PortableInterceptor.ORBInitializerClass.TSClientInit** während der ORB Intialisierung auf **org.jacorb.transaction.TransactionInitializer** gesetzt werden. Diese Klasse registriert ein **Current**-Objekt mit dem Namen **"TransactionCurrent"** als "initial reference". Praktischerweise, wurde diese Eigenschaft bereits im zur Verfügung gestellten Ant File gesetzt und kann mithilfe von "orb.resolve_initial_references" zur Verfügung gestellt werden:

```
Current ts_current = CurrentHelper.narrow(orb.resolve_initial_references("TransactionCurrent"));
```

Listing 4: Current als Intial Reference

Nachdem mithilfe von **ts_current.set_timeout(long ms)** das Transaktionslimit gesetzt wurde, kann mithilfe von **ts_current.begin()** eine implizite Transaktion begonnen werden. Danach können mehrere Operationen durchgeführt werden. Zum Schluss kann die Transaktion mit **ts_current.commit(true)** oder **ts_current.rollback()** beendet werden.

5.3.3 Aufbau

Die **deliver()**-Methode des Servers beziehungsweise der Telco und die **onSMS()**-Methode des Clients müssen innerhalb einer Transaktion durchgeführt werden. Da eine Ressource aber nur in einer Transaktion zugleich involviert sein darf, wäre es sehr unperformant wenn die jeweiligen Klassen direkt die Interfaces CosTransactions::Resource und CosTransactions::TransactionalObject implementieren würden. Daher wird ein eigenes Interface **ITransaction** in die IDL geschrieben, welches von diesen beiden Interfaces erbt. So kann gewährleistet werden dass der Client und der Server beziehungsweise die Telco Nachrichten von mehreren Transaktionen zugleich empfangen können. Dieses Interface legt außerdem eine Methode **useMessage()** fest, welches einfach eine Message als Input-Parameter verlangt und kann für Server und Client verschieden implementiert werden.

5.3.4 Client

Fuer den Client muss nun eine Implementierung **TransactionImpl** des Interfaces geschrieben werden, welche bei **useMessage()** die empfangene Message als Attribut speichert, und erst bei **commit()** tatsaechlich ausgibt:

```
public class TransactionImpl extends ITransactionPOA {

    private Message m;

    @Override
    public void useMessage(Message m) {
        this.m = m;
    }

    @Override
    public void commit() throws NotPrepared, HeuristicHazard, HeuristicMixed, HeuristicRollback {
        System.out.println("Receive message within a transaction:");
        System.out.println(TelcoHelper.formatMessage(m));
    }
}
```

Listing 5: TransactionImpl des Clients

Dafür wird im Client in der **onSMS()** Methode diese Implementierung für jede empfangene Nachricht instanziert werden. Die Message wird als Argument der useMessage() Methode von TransactionImpl übergeben, und die Instanz des **TransactionImpl**s wird mithilfe des Coordinators tatsächlich als Ressource festgelegt. Das Control Objekt wird genauso wie das Current Objekt geholt, und liefert beim Aufruf der **get_coordinator()**-Methode einen Coordniator. Danach kann eine Ressource mit **register_resource()** registriert werden.:

```
    @Override
    public boolean onSMS(Message m) {
        ITransactionPOA ti = new TransactionImpl();
        ti.useMessage(m);
        try {
            Control control = CurrentHelper.narrow(orb.resolve_initial_references("
                TransactionCurrent")).get_control();
            Coordinator coordinator = control.get_coordinator();
            coordinator.register_resource(ti._this(orb));
        } catch (InvalidName | Unavailable | Inactive ex) {
            Logger.getLogger(Client.class.getName()).log(Level.SEVERE, ex.getMessage(), ex);
            throw new TRANSACTION_ROLLEDBACK();
        }
        return true;
    }
```

Listing 6: Transaktion beim Client

5.3.5 Server

Fuer den Server muss nun eine Implementierung **TransactionImpl** des Interfaces geschrieben werden, welche vor **useMessage()** überprüft ob der User online oder offline ist. Falls der User online ist, wird kein Transaktionskontext gebraucht und die Nachricht kann ohne weiteres übermittelt werden. Falls der User doch offline ist, wird die empfangene Message in useMessage() als Attribut speichert, und erst bei commit() tatsaechlich in die Mailbox des Users geschrieben:

```
1  public class TransactionImpl extends ITransactionPOA {

       private Message m;

       @Override
6      public void useMessage(Message m) {
           this.m = m;
       }

       @Override
11     public void commit() {
           if (m != null) {
               user.addToMailbox(m);
           }
       }
16 }
```

Listing 7: TransactionImpl des Servers

Dafür wird im Server in der **deliver()** Methode diese Implementierung für jede zu sendende Nachricht instanziert werden. Die Message wird als Argument der useMessage() Methode von TransactionImpl übergeben, und die Instanz des **TransactionImpl**s wird mithilfe des Coordinators tatsächlich als Ressource festgelegt. Das Control Objekt wird genauso wie das Current Objekt geholt, und liefert beim Aufruf der **get_coordinator()**-Methode einen Coordniator. Danach kann eine Ressource mit **register_resource()** registriert werden.:

```
       @Override
       public boolean deliver(Message m) {
           try {
4              for (TelNr rnr : m.reciever) {
                   if (!rnr.prefix.equalsIgnoreCase(prefix)) {
                       continue; // other telco - ignore
                   }
                   User receiver = users.get(rnr.suffix);
9                  if (receiver != null) {
                       TransactionImpl ti = new TransactionImpl(receiver, orb);
                       if (receiver.getCallBack() != null) {
                           receiver.getCallBack().onSMS(m);
                       } else {
14                         ti.useMessage(m);
                       }
                       Control control = ts_current.get_control();
                       Coordinator coordinator = control.get_coordinator();
                       coordinator.register_resource(ti._this(orb));
19                 } else {
                       ts_current.rollback();
                       return false;
                   }
               }
24         } catch (Unavailable | Inactive | NoTransaction ex) {
               Logger.getLogger(Client.class.getName()).log(Level.SEVERE, ex.getMessage(), ex);
               throw new TRANSACTION_ROLLEDBACK();
           }
           return true;
29     }
```

Listing 8: Transaktion beim Sever

Zusätzlich wird beim Server in der send()-Methode noch die Transaktion gestartet und am Ende der send()-Methode commited:

```
@Override
    public boolean send(String suffix, String pin, Message m) {
        try {
            ts_current.set_timeout(1);
            ts_current.begin();
            try {
                User user = loginUser(suffix, pin);
                m.sender = user.getNr(); // sicherheitshalber korrekten sender setzten!
                this.deliver(m);

                ConcurrentHashMap<String, IPublicTelco> cache = new ConcurrentHashMap<>();
                for (TelNr rnr : m.reciever) {

                    String prefix = rnr.prefix;
                    if (!prefix.equals(this.prefix)) { // an eigene clients wurden schon gesendet
                        IPublicTelco server = cache.get(prefix);
                        if (server == null) {
                            server = IPublicTelcoHelper.narrow(naming.resolve_str(prefix));
                            cache.put(prefix, server);
                        }
                        server.deliver(m);
                    }
                }
                ts_current.commit(true);
                return true;
            } catch (<viele Exceptions> ex) {
                    ts_current.rollback();
                    Logger.getLogger(Server.class.getName()).log(Level.SEVERE, ex.getMessage(),
                        ex);
            }
        } catch (SubtransactionsUnavailable ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE, ex.getMessage(), ex);
        }
        return false;
    }
```

Listing 9: Transaktion Starten beim Sever

5.3.6 Explizite Transaktionen

Zusätzlich wurde noch verlangt die Methoden, die implizite Transaktionen verwenden, auch noch für die Nutzung von expliziten Transaktionen zu implementieren. Dazu wurden die sendSms()- und die onSms()-Methode des Clients, die send()-, die setORB()- und die deliver()-Methode des Servers und die useMessage()-Methode des Transaktionsfähigen Objektes neu implementiert. Der groesste Unterschied liegt in der Art wie der ORB gesetzt wird. Dabei wird nähmlich nicht das Current-Objekt aus dem Transaction-Service geholt, sondern eine TransactionFactory.

```
public void setORBExplicit(ORB orb) throws org.omg.CORBA.ORBPackage.InvalidName {
        try {
            this.orb = orb;
            NamingContextExt nc =
                    NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
            NameComponent[] name = new NameComponent[1];
            name[0] = new NameComponent("TransactionService", "service");
            this.transactionFactory =
                    TransactionFactoryHelper.narrow(nc.resolve(name));
        } catch (CannotProceed | InvalidName | NotFound ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
        }
}
```

Listing 10: Holen der TransactionFactory

Weiters muss einfach nur zum Starten der Transaktion aus der TransactionFactory ein Current-Object erzeugt werden, mit dem anschliessend die Transaktion verwaltet werden kann. Will man die Transaktion nun rollbacken oder committen, muss man zuerst aus dem Control-Objekt einen Terminator getten und kann die Transaktion mit diesem ähnlich der impliziten Transaktionen verwalten. Dies kann im folgenden Code-Snippet gesehen werden.

```
    @Override
2   public boolean sendExplicit(String suffix, String pin, Message m) throws DeliverException {
        try {
            boolean wasWorking = true;
            this.control = transactionFactory.create(20);

7           User user = checkLogin(suffix, pin);
            m.sender = user.getNr(); // sicherheitshalber korrekten sender setzten!
            wasWorking = this.deliverExplicit(m, control);

            ConcurrentHashMap<String, IPublicTelco> cache = new ConcurrentHashMap<>();
12          for (TelNr rnr : m.reciever) {
                String prefix = rnr.prefix;
                if (!prefix.equals(this.prefix)) { // an eigene clients wurden schon gesendet
                    IPublicTelco server = cache.get(prefix);
                    if (server == null) {
17                      server = IPublicTelcoHelper.narrow(naming.resolve_str(prefix));
                        cache.put(prefix, server);
                        System.out.println("Contacted telco \"<" + server.getName() + ">\" (<" +
                            prefix + ">) successfully.");
                    }
                    if(wasWorking) {
22                      wasWorking = server.deliverExplicit(m, control);
                    }
                }
            }
            if(wasWorking) {
27              control.get_terminator().commit(true);
            }
            return true;
        } catch (org.omg.CORBA.TRANSIENT | org.omg.CORBA.COMM_FAILURE | NotFound | CannotProceed
            | InvalidName | InvalidLoginException | DeliverException | HeuristicHazard |
            HeuristicMixed ex) {
            try {
32              control.get_terminator().rollback();
                Logger.getLogger(Server.class.getName()).log(Level.SEVERE, ex.getMessage(), ex);
            } catch (Unavailable ex1) {
                Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex1);
            }
37      } catch (Unavailable ex) {
            throw new DeliverException();
        }
        return false;
    }
```

Listing 11: Explizite send-Methode im Server

Sonst ist die Handhabung bei expliziten Transaktionen sehr ähnlich zur der bei impliziten Transaktionen. Lediglich die Methode des transaktionsfähigen Objektes muss noch mit einer zusätzlichen Übergabe des Control-Objektes vesehen werden.

6 Testbericht

6.1 SystemTest

Es wurden die Methoden des Clients mittels einem Verbindungsaufbau zum TMobile-Server getestet. Dazu wurde das test-client Target im Ant File wie folgt umgeschrieben:

```
<target name="test-client" depends="compile" description="Run all tests for the client
    implementation.">
    <mkdir dir="${build.dir}"/>
    <javac srcdir="${test.dir}" destdir="${build.dir}" includeantruntime="true">
        <classpath refid="testing.classpath"/>
    </javac>
    <mkdir dir="tmp/rawtestoutput"/>
    <parallel>
        <daemons>
            <sequential>
                <ant antfile="build.xml" inheritall="false" target="run-ns"/>
            </sequential>
            <sequential>
                <sleep seconds="5"/>
                <ant antfile="build.xml" inheritall="false" target="run-ts"/>
            </sequential>
            <sequential>
                <sleep seconds="10"/>
                <ant antfile="build.xml" inheritall="false" target="run-server">
                    <property name="props" value="tmobile"/>
                </ant>
            </sequential>
        </daemons>
        <sequential>
            <sleep seconds="15"/>
            <junit haltonfailure="false" printsummary="true" fork="no">
                <sysproperty key="org.omg.CORBA.ORBClass" value="org.jacorb.orb.ORB"/>
                <sysproperty key="org.omg.CORBA.ORBSingletonClass" value="org.jacorb.orb.
                    ORBSingleton"/>
                <sysproperty key="ORBInitRef.NameService" value="file://${ns.dir.location}/NS_REF
                    "/>
                <sysproperty key="java.util.logging.config.file" value="${build.dir}/logging.
                    properties"/>
                <sysproperty key="org.omg.PortableInterceptor.ORBInitializerClass.TSClientInit"
                             value="org.jacorb.transaction.TransactionInitializer"/>
                <classpath path="${build.dir}">
                    <path refid="testing.classpath" />
                </classpath>

                <batchtest todir="tmp/rawtestoutput">
                    <fileset dir="test"/>
                    <formatter type="plain"/>
                </batchtest>
            </junit>
        </sequential>
    </parallel>
</target>
```

Listing 12: Test-Client Target des Ant Files

Hier werden also der Namesever, das TransactionService und der T-Mobile Server automatisch im Ant gestartet. Außerdem werden noch die gebrauchten initial references gesetzt.

Die einzelnen Testmethoden testen den Verbindungsaufbau des Clients zum Server, den Login des Clients beim Server, die Abmeldung des Clients beim Server, das Senden einer Nachricht und das fehlgeschlagene Senden einer Nachricht:

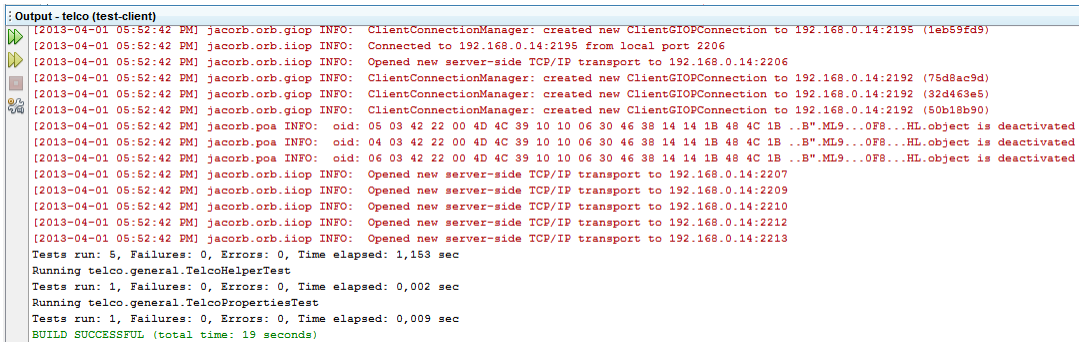

Abbildung 7: Testbericht des ClientTests

Literatur

[1] Michael Borko. Task08 - rmi messenger, 2012. `http://elearning.tgm.ac.at/mod/assign/view.php?id=11242`.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings