INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Integration of the CTTN system in Java

Julius Benkert

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach
Betreuer: Prof. Dr. Hans Jürgen Ohlbach

Abgabetermin: 27. Februar 2006

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 27. Februar 2006

.....

(Unterschrift des Kandidaten)

Abstract

The CTTN system offers powerful processing capabilities for various kinds of temporal notions. However, its core is written in the C++, so various binary packages would have to be created in order to support different system types. Additionally, each target device would require an installation of the software before it is able to process CTTN requests itself. In the beginning of the ubiquitous computing era, this approach is considered to be rather outdated. Especially mobile devices are often unable to hold large amounts of data and to process complex computations.

The thesis at hand supersedes the classical distribution by providing alternative ways to access CTTN's abilities. Communication with the C++ core is thereby realised using a plaintext socket connection. Newly created server interfaces then enable clients to create remote objects and invoke methods on them while basically just waiting for the results. First of all, an RMI interface is provided for serving Java clients exclusively. However, most contemporary computing environments use a mixture of programming languages, so setting up a server for only a single one is mostly unsatisfactory. In the last 15 year, the CORBA standard has proven that it is able to provide similar services to a variety of languages. The therefore introduced second interface to CTTN offers CORBA connectivity between clients and the server. As an alternative, an additionally created web service represents the third way to access CTTN remotely. In contrast to its rivals, this interface employs the completely XML-based SOAP protocol and thus can be used by any remote system that at least owns an XML parser.

Today's computing environments are changing rapidly, so fixing the way of access to CTTN to one specific interface is often undesirable. The newly created CTTNUniversalClient addresses this issue for Java and allows seamless migration between different underlying architectures. By using reflection, it provides a common API for all available remote interfaces.

Contents

1 Introduction

The capabilities of the default Java packages addressing date and time handling are currently very limited. Several classes in *java.util* combined with *java.text.DateFormat* offer the possibility to parse and format crisp dates. Additionally, the *Calendar* class provides some basic calculation and comparison capabilities for these points in time.

For handling more complex temporal objects and tasks, the CTTN system is able to offer a powerful alternative. To be exact, the "CTTN-system is a computer program which provides advanced processing of temporal notions. The basic data structures of the CTTN-system are time points, crisp and fuzzy time intervals, labelled partitionings of the time line, durations, and calendar systems. The labelled partitionings are used to model periodic temporal notions, quite regular ones like years, months etc., partially regular ones like timetables, but also very irregular ones like, for example, dates of a conference series. These data structures can be used in the temporal specification language GeTS (GeoTemporal Specifications). GeTS is a functional specification and programming language with a number of built-in constructs for specifying customised temporal notions."[2]

Right now, several C++ libraries form the core of the CTTN system by offering the specified processing capabilities. Access from remote hosts is so far solely provided via an attached socket server. The employed protocol for this connection type is based on a plaintext command set. A major downside of this way to connect is the loss of object orientation. In consequence, all programming languages following this paradigm can't make use of the immanent class structure of the C++ core system. The master thesis at hand addresses this deficit and gives Java clients object-based access to the CTTN system.

Instead of mapping calls to their socket protocol counterparts in the client, an additional hop is introduced. The intermediate station accesses the CTTN core system via the socket protocol and maps the calls to a Java object structure. Access for clients is then offered over distributed object systems. The advantage of this approach is that clients will never have to deal with the socket-specific protocol. They just obtain remote object references from the interface server and are able to invoke methods without the need to know about details of the internal processes. Pure Java clients can use CTTN over a Remote Method Invokation (RMI) interface that has been created as part of this work.

Figure 1: Abstract overview of the interaction between CTTN client, server interface and core instances. Client applications are connected by either RMI, CORBA or SOAP. Each remote session object owns a separate SocketClient object with its specific socket connection to the core system.

For other object-oriented programming languages, the use of RMI is not an option since it is exclusively designed to be used with Java. For that reason, an aditional interface is provided that uses the Common Object Request Broker Architecture. CORBA is supported by a wide range of programming languages, so the second CTTN interface enables non-Java clients to use its capabilities in an object-oriented way like RMI clients do. For the case that CORBA is also unavailable in the client's programming language, this work also introduces a web service as third interface. For consumption, clients can utilise frameworks to simplify programming tasks. However, the web service interface could be used with solely a XML parser available and thus should finally allow almost every client to connect to the CTTN system.

Although Java has its own way to offer and use remote objects over RMI, client Java classes are also provided for the CORBA and the web service interfaces. In many environments, a variety of programming languages is used additionally. Thus, another interface than RMI might be used for all programming languages in a given environment and should therefore be supported by the CTTN Java client implementation. It is inevitable that the provided classes for each remote interface have a different API. To hide these differences from custom client applications, an additional abstraction layer has been added to the Java client implementation. By using reflection, the so-called UniversalClient offers CTTN's RMI API for all available remote interfaces. This additional piece of software allows client developers to access the CTTN system over one common interface while still being able to exchange the utilised remote invokation architecture without modifications to their software.

1.1 Aims of the implementation

The CTTN core system is currently licensed under the GNU General Public License Version 2 or later. It is therefore open source and available free of charge. The overall state of the whole CTTN software should remain unaffected by the ad-

ditional remote interface components. The additionally developed software that is part of this diploma thesis is hence available under the same terms. To prevent any limitations to the use of CTTN, open source software has been generally preferred. Whenever something argued against this type of software, replacements have been selected that are at least distributed freely at no charge.

As far as the client implementation is concerned, CTTN is supposed to be a programming language extension that can be seamlessly integrated in almost every present environment. It would be mostly unrealistic to expect that client systems are especially prepared just for the use of CTTN. For that reason, the expectations regarding the up-to-dateness of the client's software components is reduced to the lowest reasonable level for all interface implementations. Hardware requirements should also be as low as possible to allow using the CTTN system on devices with very limited storage, network or processing capacities. These last mentioned restrictions are especially encountered on the emerging field of mobile devices.

The situation for the server part is rather different. The according implementation will be installed and run on comparatively few hosts. Since the target systems will be mainly servers, extended knowledge and efforts regarding the installation of software can be expected from the system administrators. These increased expectations have been defined to allow the newest available technologies to be used during the initial and furthermore development of the server implementations. The overall emphasis of this part is clearly put on ease of development, robustness of operation and portability of the offered interfaces. The last point is especially relevant for the design of the web service. Details on this topic are provided later in the according server side documentation. Necessities regarding the hardware equipment shouldn't be an issue on current server systems. They are mostly determined by the underlying software that is needed to provide the interfaces themselves. Since the core system of CTTN didn't reach its final functional range yet, tests regarding the throughput and resource usage of CTTN's remote interfaces wouldn't be meaningful and have therefore been skipped. The server side implementation of interfaces is anyway targeted on an economical use of resources.

Finally, the overall speed of the CTTN system is assumed to be one of the most important factors. This consideration has played a major role during the design and writing of the implementation. It has also been of particular interest for the

selection of additional software for the CTTN web service. Again, details on this point can be found in the server side web service documentation.

1.2 Basic structure

The thesis and the correspondent implementation is mainly split up in a server and a client side part. This separation has been introduced, since many files are used exclusively to either provide or consume the CTTN services. Besides a improved overall clarity, this design facilitates the generation of packages and documentation limited to only one side. Another positive effect thereof is, that client packages don't contain unnecessary files and thus don't waste storage resources on the target device. Resources used by both parts are stored in shared folders, which are automatically included from both sides. Nonetheless, package and documentation versions enclosing the whole project are created as well. An additional folder *bin/local* is included in the *CLASSPATH*, but not in the generated packages. It currently contains only the specific configuration of the local system.

The server side part of the documentation at hand also contains the necessary steps to create the client side bindings. This might sound a litte bit inconsistent, but is necessary for two reasons. First of all, the automatic generation of required RMI files is based on the server implementation and thus can't be performed by clients. The documentation of all server interface implementations also describes how to test the provided interfaces. The presence of client side bindings is necessary for the therefore executed classes.

The following tree sketches the basic layout of the project's folder structure. This overview is also contained in the file *structure.txt* in the project root.

```
cttn                Project root
 |-- src              Java sources
 |    |-- server       Required only for the server side
 |    |-- client       Required only for the client side
 |    |-- shared       Required for both sides
 |-- bin              Java binaries
 |    |-- server       Required only for the server side
 |    |-- client       Required only for the client side
```

```
|    |-- shared        Required for both sides
|    |-- local         Included in CLASSPATH but not in packages
|-- api               API Documentation (by javadoc)
|    |-- server        Only for the server side and shared sources
|    |-- client        Only for the client side and shared sources
|    |-- full          For all available sources
|-- packages          Binary packages (full/server/client) plus Axis2 aar
|-- policies          Security policies for RMI
|-- deployment        Deployment stuff for web service (SOAP)
|-- var               Various information on and from server runlevel scripts
     |-- log          Log files of server applications
     |-- pid          Pid files for started daemons/applications
```

1.3 Common prerequisites

The whole implementation described within this document is based on Sun's "Java 2 Platform, Standard Edition" SDK in version 5.0, which is available for download at http://java.sun.com/j2se/. For pure clients, a J2SE Runtime Environment as of version 1.2 should be sufficient to directly access all interfaces. If the UniversalClient is about to be used, the lowest supported Java version number increases to 1.3. However, the lowest tested runtime environment version is 1.4.2, so the use of at least this release is recommended for client side use. The preferred use of open source software has been deferred for this selection since Sun's Java implementation can still be considered as reference implementation. It is already present on many target systems and has extensively proven its compatibility with the software components that are used on the server side. However, the use of a client side alternative should be possible without modifications. A few details of the server implementation are set specifically for the Sun implementation and probably need to be changed before a substitute can be used.

Once Java is installed properly, clients have to include the archive *cttnClient.jar* in their *CLASSPATH* to gain access to the CTTN system. There is also a server side counterpart called *cttnServer.jar*. Alternatively, the archive *cttnFull.jar* may be included to support both modes of operation.

2 Resources required for both sides

As already mentioned, some components are used by clients as well as by servers. Among them are the interface and stub binaries used for RMI. A helper class and the IDL file defining the CORBA interfaces are also part of the shared sources. Complete descriptions of these files are given in the according sections. Besides those interface-specific files, two classes are used by multiple interfaces.

2.1 The *CTTNException* class

The first of them is the exception class *de.lmu.ifi.CTTN.CTTNException*. Since no detailed exception model has been defined for CTTN's socket interface yet, all kinds of CTTN-related errors cause a *CTTNException* to be thrown. A descriptive message may be included upon construction. Since exceptions are serialized and transmitted to the client, almost all interfaces are able to use this exception class without modifications. Only CORBA insists on a special inheritance for all exceptions, hence the general *CTTNException* can't be used there. More on this issue can be found in the server side CORBA part.

2.2 Reading configuration values via *CTTNConfig*

The creation of the second class evolved from the fact, that many configuration properties need to be accessed at various points of the server implementation. Besides that, it is desirable to have a place where clients can be equipped with default values e.g. for server addresses to use. This might be of special interest for centrally administered environments, since the Java environment can be preconfigured to use a specific CTTN server. In addition, custom software using the CTTN system can be deployed without defining a default server to be used and thus without exposing a single host to all arising requests.

Unlike encountered in the configuration of classical linux services, Java applications should try to avoid the use of absolute paths for their configuration files. A preferred way to solve this issue is to use configuration files with a *properties* extension that are placed within the *CLASSPATH*. The priority of a configuration file is then determined by its path's position in the *CLASSPATH*, so the first

matching file will be used. The so-called *ResourceBundle*s just contain property-value assignments. A value can thus be accessed simply by its property name. The respective file for the configuration of the CTTN interfaces needs to be placed in the *de.lmu.ifi.CTTN* package and has to be named *config.properties*. A single file for the server and the client settings was chosen due to the fact, that some values are of use for both sides. A complete list of available configuration entries is available in appendix A.

Instead of directly accessing the *ResourceBundle* from various points in the code, the intermediate class *CTTNConfig* out of the same package is used. It basically wraps the functionality of a *ResourceBundle* but adds some convenience methods. The additional methods already return the desired object type like for instance an *URI* or only a part of the value. In consequence, they save plenty of coding efforts and improve the resulting code's clarity. This also applies to potentially thrown exceptions which are automatically mapped to CTTN's exception model. Finally, a *main* method has been added to give non-Java programs access to the configuration. This feature especially comes in handy when starting required daemons and for setting *System* properties upon the invokation of Java applications. Multiple arguments may be used to retrieve several properties at once. The results will then be separated by a single space character. Since it is often necessary to obtain only one component of an *URI* property value, a special syntax has been defined for this purpose. The property is therefore still accessed by its name, but an additional colon followed by the name of the desired part reduces the returned value accordingly.

A last note concerns the point in time when the configuration *ResourceBundle* is read. The according property is defined to be *static*, so the configuration file could also be read in a *static* block. An implication of this approach would be, that a nonexistent *config.properties* file would cause a *MissingResourceException* to be thrown. This behaviour is unwanted, since many of the property values accessible via *CTTNConfig* are mainly intended to be default values. If other settings override them, there is no need to complain about missing files. This is why the *resourceBundle* class property will not be assigned until the first property value is requested. In other words, the interface implementations work well without the presence of a configuration file as long as no calls to *CTTNConfig*'s methods take place.

3 Serverside design and implementation

First of all, it has to be pointed out, that the server part of the CTTN interface project does not directly implement the offered capabilities. All computations are performed by the core of the CTTN system, which is as already mentioned written in C++.

3.1 Communication with the CTTN core

For communication with the backend, two different approaches have been evaluated. Since Java is generally able to interact with C++ libraries by using the Java Native Interface (JNI), this has been the first regarded solution. The most remarkable feature thereof would be the possibility to use C++ object structures directly in Java. The according steps to create a JNI-enabled library are also quite simple. Nonetheless, this approach has several negative implications. The most obvious one is, that the CTTN core system library and the server part of this implementation have to run on the same server. This is not necessarily a problem, but reduces the overall flexibility. Furthermore, the core libraries have to be available for the specific server system as well as the required components of the remote interface implementations. More downsides of JNI arise from the current design of the CTTN core system. Several required features are part of additional software and thus aren't included in the libraries. Adding them to the libraries would require serious modifications and in fact this isn't desired.

The alternative solution makes use of a plaintext protocol over a socket connection. A major downside of this approach is clearly the loss of object structures in conjunction with the employed protocol. Anyway, the socket communication offers some considerable strengths. First of all, there is no need to run the core system on the same host as the remote interface implementations. This circumstance can be used to build separate backend servers which are not directly exposed to potentially harmful clients. In addition, more sophisticated features like load balancing for example can be realised for all remote interfaces at once just by switching backend server adresses. This can also be used to provide an automatic failover for the backend computation.

The higher overall flexibility of the socket approach combined with the downsides of a JNI solution finally lead to the selection of the second alternative. An

according server implementation is already part of the CTTN core system, so this solution is ready for use without modifications yet.

3.1.1 The regular socket client

To get back the object orientation which was lost during transmission, a set of classes has to map elements of the plaintext protocol to according objects. Analogously, calls to Java CTTN objects have to be translated in their plaintext string representations. Both tasks are performed by the object classes contained in the *de.lmu.ifi.CTTN.Socket* package and its subpackages.

The initially required class for using the CTTN core system is called *CTTNSessionImpl*. Once a new object is instantiated, a socket connection to the backend server is automatically created. The server's address can be specified by an optional *URI* parameter upon construction. If the parameter is omitted, the values to use are obtained from the *System* properties *de.lmu.ifi.CTTN.Socket.host* and *de.lmu.ifi.CTTN.Socket.port*. If those aren't supplied either, the *socketURI* property is read from *CTTNConfig*. Upon a successful connection, the welcome line sent by the server is parsed for the version strings of the CTTN server software and the used protocol. From now on, the *CTTNSessionImpl* object can be used to execute requests. All communication tasks following an request-response-scheme use the public function *execRequest*. This function sends the request to the server, returns the response or alternatively throws an exception if necessary. To terminate a session, the client should call the *end* method on the *CTTNSessionImpl* object to close the connection to the server.

If one of the tasks mentioned in the last paragraph fails, a *CTTNException* from the *de.lmu.ifi.CTTN* package will be thrown with a descriptive error message. The socket client is meant to be just a backend for the remote interfaces. That's why even transport errors are declared to be CTTN-internal and lead to a *CTTNException* as well. In contrast, the clients of the RMI, CORBA and SOAP interfaces distinguish between transport errors and CTTN-related exceptions.

A basic requirement for the use of CTTN is that all sessions have to be kept strictly separated. Otherwise, a client would be able to modify the values of other sessions. The used socket approach automatically fulfils this need. The socket server of the CTTN core system creates a new, entirely separated environment for every connection. Since each *CTTNSessionImpl* maintains its own connection

to the backend server, the clients' environments are also isolated from each other in the respective Java implementation.

The following sample code fragment starts a *CTTNSessionImpl*, instantiates a new *CTTNInterval* by parsing an interval string and then prints CTTN's human readable representation of the resulting interval.

```
import java.net.URI;
import de.lmu.ifi.CTTN.CTTNException;
import de.lmu.ifi.CTTN.Socket.CTTNSessionImpl;
import de.lmu.ifi.CTTN.Socket.FuTI.CTTNIntervalImpl;
try {
  /* Build the server's URI */
  URI uri = new URI("telnet://localhost:1953/");
  /* Start the session with explicit specification of
     the server */
  CTTNSessionImpl session = new CTTNSessionImpl(uri);
  /* Parse an interval representing january 2006 */
  CTTNIntervalImpl interval = session.parseInterval("
     2006/01");
  /* Print out some properties of the interval */
  System.out.println("Interval␣"+interval.getId()+"␣
     covers␣"+interval.date()+".");
  /* Close the socket connection to the server */
  session.end();
} catch (java.net.URISyntaxException e) {
  System.err.println("Parsing␣the␣URI␣string␣failed.");
} catch (CTTNException e) {
  e.printStackTrace();
}
```

Listing 1: This starts a session, parses an interval and prints its human readable representation via the socket client.

A working configuration and a CTTN socket server listening on port 1953 presumed, running listing 1 yields the output "Interval 0 covers [2006 2006/2[."

The complete description of the socket client's abilities is available via its API. Background information regarding the structure and logic of the CTTN sys-

tem can be obtained from the papers on CTTN[2] and it's modules. The currently evolving extensions are FuTIRe[3] for fuzzy time intervals and relations, PartLib[5] for labelled partitionings and the language for geo-temporal notions called GeTS[4]. The most up-to-date documentation versions are available on the website of the CTTN project at http://www.pms.ifi.lmu.de/CTTN/.

3.1.2 An extended socket client class

Finally, the class *CTTNExtendedSessionImpl* contained in the socket client's package needs to be mentioned here. It extends the regular *CTTNSessionImpl* class by a secure session identifier contained in the object property *id*. An additional method *generateId*, which sets this identifier automatically, is invoked during the construction of the object. The identifier itself is created by the concatenation of an incremental counter value, an underscore and a MD5 hash value. The plaintext value for the hash is composed of the configuration value *sessionSalt*, the incremental counter value and the timestamp measured in the highest available resolution. The counter component garants that the hash values of two exactly simultaneously generated session are different. By utilising a timestamp with the highest available precision, a possible attacker can hardly be sure about the exact point in time of the identifier generation. In addition, the value of *sessionSalt* has to be set by the server administrator to a secret, strong and random string. The combination of those three components ensures an unguessable plaintext for every session, thus preventing chosen plaintext attacks on the session identifiers. Another additional feature of the extended socket client class addresses automatic expiration of session. The object property *lastUsage* in combination with the methods *updateLastUsage* and *isPurgeable* is used to determine, if a session is still valid or if it has expired. This ability is only necessary for the web service implementation. The feature is nonetheless added here to circumvent the introduction of another inheritance level for an according web service socket client.

3.2 Provided shell and init scripts

The successive RMI, CORBA and web service subsections also contain instructions how to bring up the server side remote interfaces. Alternatively, various

scripts in the project's base directory automate the necessary preparation of the environment as well as the service management. They are currently intended to be used for developing and testing the interfaces. Therefore it is recommendable to read and understand the function of the shell scripts and adapt them wherever necessary before using them in productive environments. To avoid persistent errors during development, startup and shutdown of the CTTN's interfaces always cause a restart of the according daemons. This behaviour is probably unwanted in environments with multiple services offered by a single daemon.

To adapt the provided scripts to a computer's environment, the file *setPaths* should be adjusted first. It contains the paths to the software used to realise the interfaces and will probably require some modifications to fit another system's environment. Anyway, the current Java package for CTTN already contains all necessary software except the Java SDK in the directory *software* and should run quite out-of-the-box. The purpose of the *setPaths* script is to ensure that all necessary environment variables are properly set and that the *CLASSPATH* contains all Java packages and directories used by the interfaces' implementations.

A special note concerns the package *libgcj* that is deployed with SuSE LINUX Professional 9.3 and possibly other distributions. Besides some libraries necessary for the GNU java compiler gcj, this particular package contains special versions of the RMI components *rmic* and *rmiregistry*. The default path settings prefer those binaries over Sun's corresponding versions. Unfortunately, the RMI component of CTTN is incompatible with the mixed up default installation of Sun and GNU java implementations. To avoid problems with binaries from different products, it is necessary to ensure that either the environment is properly prepared or all relevant binaries are called with their absolute paths. The scripts provided with this package already address this issue by using the second possibility.

Besides the described customization, no further actions should be necessary to run the server components of CTTN's RMI, CORBA and SOAP interfaces. All init scripts obtain their configuration from the main method of the *CTTNConfig* class documented earlier in this thesis. The only exception to this statement is the port of the application server used for SOAP. If necessary, it needs to be adjusted in the configuration files of this specific server.

The script files *rmi*, *corba* and *soap* are init scripts for the different interface implementations. The tasks *start*, *stop* and *status* are currently implemented. As mentioned above, they also control the corresponding daemons. In case of RMI

for instance, the init script starts the *rmiregistry* and the *rmid*. Afterwards, the actual remote object is initialised and registered. The *stop* tasks shut down the started processes in inverse order. A call to the *status* task checks if the processes are running and prints out the resolved states. The file *cttn* is a convenience script that executes a task on all interfaces.

3.3 The RMI interface

Java's Remote Method Invocation consists of a set of tools and a proprietary transport protocol. It serves the purpose of using remote program-level objects in a Java client application. First of all, besides many strengths, one of the major weaknesses of RMI is its lacking interoperability with other programming languages. But at the same time, this limitation is one of it's main advantages. Because there is no need to map the objects' interfaces to language-independant ones, they can be defined directly as Java interfaces with only a few additional requirements. In consequence, maintainance remains very straightforward and the overhead of the remote method invocation can be comparatively smaller than that of portable architectures.

The first service encountered when using a remote object over RMI is the naming service. Its purpose is to map a name, generally a plaintext string, to a remote object reference. It is worth mentioning, that this service does not necessarily need to reside on the same physical host or under the same network address as the remote objects it handles. The realisation of the service is called *rmiregistry* and is already contained in Sun's Java SDK. Once the daemon is started, it binds to a tcp port and waits for requests. By default, the *rmiregistry* binds to port 1099. If desired, this can easily be changed by supplying another port number as first parameter. Server applications can now bind names to remote objects via this registry. Clients can then contact the registry and look up remote objects registered this way by just providing the objects' respective names. In general, the following line is completely sufficient to start a *rmiregistry* in the background:

```
rmiregistry &
```

3.3.1 Defining the remote interfaces

The first step towards remotely available objects is the definition of their interfaces usable for remote calls. Remote interfaces almost look like locally used ones and vary only in a few points. One difference is, that an interface of a remote object must extend *java.rmi.Remote*. This interface doesn't declare any methods, it is just an identifier marking a remotely accessible object. Unlike objects implementing the *Serializable* interface, those implementing *Remote* are never transmitted to the client in a serialized form. The clients will instead receive a reference to them. The only other difference is, that all methods of a remote interface have to declare a *java.rmi.RemoteException* to be thrown. This exception covers all errors immanent to the system of remote method invocation. Some very likely reasons for such errors are not — or no longer existent — objects as well as network related issues.

Figure 2: This sequence diagramm shows the creation, use and termination of a remote client session. The creation of objects within a session has been omitted for clarity. The software parts can be spread over up to three different hosts, namely the client host, the interface server and the core server doing the processing. Clients are able to access the CTTN interface server either over RMI, CORBA or SOAP. The communication to the core system is realised via a socket connection.

The interfaces defined for CTTN's RMI support are placed in the client project's package *de.lmu.ifi.CTTN.RMI* and below. Their declared functionality is equal to those of the socket client's classes, they are just slightly modified to meet the additional requirements for remote interfaces mentioned above. Listing 2 shows the remote interface *CTTNSessionManager*, which provides just one method to start new sessions. This additional interface is required to meet the demand of isolated sessions. If a *CTTNSessionImpl* object would be directly bound to a name in the *rmiregistry*, all client look-ups for this name would yield the same session object with the same assigned socket connection. In consequence, a single environment would be shared among all RMI clients and would thus violate the demand. As one might notice, it is not obvious, whether the return value sent to the client will only be a substitute for a remote object or a serialized representation of the object itself. Which behaviour is finally encountered is only visible in returned interface's declaration, depending on whether it extends *java.rmi.Remote* or not.

```
public interface CTTNSessionManager extends java.rmi.
   Remote {
  public de.lmu.ifi.CTTN.RMI.CTTNSession startSession
     () throws java.rmi.RemoteException, de.lmu.ifi.
     CTTN.CTTNException;
}
```

Listing 2: The remote interface *CTTNSessionManager* declares the remotely usable *startSession* method.

3.3.2 Creation of the server classes

In step number two, the server classes are created. Of course, they have to implement the remote interfaces defined in the previous step. Besides that, the implementation itself is almost identical to the one of a regular local object. An exemption to this statement is necessary for the creation of constructors. Since the objects need to be accessible to clients later on, they have to be exposed in some way. RMI accomplishes this task by binding them to a TCP port. The port doesn't need to be assigned exclusively, so a lot of different remote objects can share a single port. The process of port allocation can be handed over to the class *java.rmi.server.UnicastRemoteObject*. The first way to do this is simply by extending the class. If the part of the superclass is already taken, an alternative is calling *UnicastRemoteObject*'s static method *exportObject* in the constructor of the server class. Both approaches yield exactly the same result. The process of port allocation needs some attention anyway. In many environments, access to most ports from remote is blocked or cloaked by default. *UnicastRemoteObject*'s standard behaviour of binding to a virtually randomly chosen port is hence problematic. The *CTTNConfig* property *rmiObjectPort* resolves this problem by allowing to fix the port number to a specific value. All constructors follow this specification by calling a version of *exportObject* with the defined port. If the property is unset or the value is set to zero, the original behaviour is restored. Altogether, the functionality offered by *UnicastRemoteObject* is sufficient for objects without special requirements regarding error recovery. Therefore, all remote object implementations except the *CTTNSessionManagerImpl* use this base class.

The implementation of *CTTNSessionManager* has been realised in a different way

because of an unwanted effect associated with the use of *UnicastRemoteObject*. Whenever a uncaught *RuntimeException* or anything alike causes a termination of the remote object, all subsequent requests would fail because of the dead remote object. The only way to get the service back up would be a manual restart of the initial remote object's registration procedure. This would make a new remote object available under the chosen name. This whole problem can be avoided by using Java's activation system based on the package *java.rmi.activation*. When a remote class uses activation, its remote instances are managed by a special daemon called *rmid*. The daemon can start virtual machines and remote objects as necessary on client requests. A so-called *ActivationGroup* can contain multiple activatable remote objects within a shared environment. *ActivationGroup*s are responsible for creating new instances of member objects. For each *ActivationGroup*, the properties of the associated virtual machine need to be set by an appropriate description upon registration of the group. The necessary class for these settings is *ActivationGroupDesc*. The registration with the activation system returns an *ActivationGroupID* which unambigously identifies the group. Before a remote object can be activated within this group, several properties need to be set on the object level as well. The class for this purpose is named *ActivationDesc* analogously. In detail, the information necessary upon construction is the *ActivationGroupID*, the object's full class name and the location, where the code can be obtained from. It is possible to initialise the object with predefined data by setting a fourth parameter accordingly. Since the CTTN implementation doesn't need this feature, the fourth parameter is simply set to *null*. With the *ActivationDesc* object at hand, a consecutive call to the *register* method of the *Activatable* class enables the dynamic initialisation of the remote object. The stub which is returned by the *register* method can then be bound to the *rmiregistry* with a custom name. Whenever a client requests an object with the chosen name from the registry, the activation system can return either an existing instance of the remote object or a newly created one.

3.3.3 Defining security policies

Before the *rmid* can be started, it is necessary to have a look at the security management system it uses. Allowing remote access to local objects is always connected with certain threats to the security of the local system. These include execution of arbitrary code, theft or modification of data, abuse of bandwidth

and many more. To prevent those unwanted incidents, Sun's implementation of the *rmid* requires the specification of a policy file for the rmid. The policy thereby is defined by a set of permissions, which can give applications rights in various categories like *File*, *Socket* and *Net*. Each of these categories is represented by its own subclass of *java.security.Permission*. In conjunction with Sun's *rmid*, the most commonly used permission is *com.sun.rmi.rmid.ExecOptionPermission*. By using this permission, it is possible to allow the setting of *System* properties for *ActivationGroup*s to values matching the defined patterns.

```
grant {
    // Allow activation groups to set and use certain system properties
    permission com.sun.rmi.rmid.ExecOptionPermission "-Djava.security.polic
    permission com.sun.rmi.rmid.ExecOptionPermission "-Djava.class.path=*"
}
```

Example TODO: An exemplary excerpt from the *rmid*'s policy file. The first permission allows setting *java.security.policy* to the value of the property *de.lmu.ifi.CTTN.RMI.policyFile* in the *rmid*'s virtual machine. The second permission enables arbitrary classpaths values to be used for *ActivationGroup*s.

Besides the *rmid*'s policy file, a second policy file specifies the allowed operations of the activatable object's *ActivationGroup*. For the current implementation, only limited network access has to be granted to the server objects. Since local objects on the server are exposed for client use by binding them to an anonymous TCP port, this has to be allowed explicitly. The second thing a server object must be able to do is communicating with the CTTN backend to process the client's requests. For both permits, a *java.net.SocketPermission* must be issued. The first parameter of the *SocketPermission* specifies the endpoint, the second one indicates the allowed actions. The fact that the so-called *codeBase*, the place where all object classes are stored, is known, allows a restriction of the permissions to only those objects contained in there.

```
grant codeBase "${de.lmu.ifi.CTTN.codeBase}" {
    permission java.net.SocketPermission "*:1024-","accept";
    permission java.net.SocketPermission "${de.lmu.ifi.CTTN.Socket.host}:$
};
```

Example TODO: The first *SocketPermission* of this policy file allows all objects running within CTTN's RMI *ActivationGroup* to bind to arbitrary local ports on as well arbitrary local interfaces. The second permission approves communication with the CTTN backend trough a socket connection to the host and port specified in the *System* properties *de.lmu.ifi.CTTN.Socket.host* and *-.port*.

3.3.4 Starting the RMI daemon

With both policy files at hand, the *rmid* can now be started. In order to know, where the policy files reside, two *System* properties need to be set on startup. The property *java.security.policy* has to contain the absolute path of the *rmid* policy file. The path to the *ActivationGroup*'s policy file is specified via *de.lmu.ifi.CTTN.RMI.policyFile*. This property is first used by the *rmid* policy file to prevent other *ActivationGroups* from setting their policy file path to an arbitrary value. Another useful property affects the hostname that is associated with remote stubs of locally generated objects. On servers with multiple network interfaces or addresses, the autoselected hostname might point to an interface unreachable for potential clients. On some systems, the chosen hostname can even be just wrong. Setting *java.rmi.server.hostname* overrides the autoselected value, so the administrator can adapt the value to the server's specific configuration. An overview of all available properties regarding RMI can be found in the Sun's Java 5.0 documentation at http://java.sun.com/j2se/1.5.0/docs/guide/rmi/javarmiproperties.html. The properties defined at startup are prefixed with *-J*. This means that they are intended to be set on the java interpreter running *rmid*. In contrast, a *-C* prefix would cause the properties to be set on the virtual machines of the *ActivationGroup*s. The optional parameter *-port* may be used to bind *rmid* to another port than 1098.

```
rmid -J-Djava.security.policy=/rmi/policy/path
-J-Dde.lmu.ifi.CTTN.RMI.policyFile=/activationGroup/policy/path
-J-Djava.rmi.server.hostname=cttn.pms.ifi.lmu.de
-port 12345
```

3.3.5 Generation of stub classes

Until Java version 5.0, the next mandatory step towards a working remote method invocation had been the manual pregeneration of the clients' bindings to the remote objects. The so called stubs produced that way acted as local proxies. Their purpose was to transparently hide the communication with the server from the client, hence allowing the client to handle the objects like local ones. The pregeneration itself had to be done by a tool called *rmic*. The resulting stubs had to be made available to the client in addition to the remote interfaces. Otherwise, the client would get a *ClassNotFoundException* upon deserialization of the remote object's stub. Nonetheless, a server application running on Java 5.0 still causes an attempt to load precomplied stubs for remote objects. But instead of throwing an exception if this loading fails, the remote objects are simply exported using a dynamic proxy. By using this newer approach, it is no longer necessary to precompile stubs for every remote object. Anyway, since the resulting remote object reference appear as stubs in one case or as proxy objects in the other, the client has to use the same wrapper as the server. The problem of this correlation is, that Java versions 1.4.x and below don't contain the required *Proxy* class yet. In consequence, the use of the RMI interface based on the new approach would require at least version 5.0 clients. One of the defined aims of the implementation at hand is good portability on the client side. Because Java version 1.4 is still very widely spread, it should consequently be supported. That's why — for now — the stubs are still generated and included in both the server and the client packages. When the use of versions lower than 5.0 vanishes, the technique used for hiding communication can be easily changed on the server side. The clients will then automatically instantiate a Proxy object instead of the corresponding stub.

The following example call to *rmic* generates a stub for the remote object class *de.lmu.ifi.CTTN.RMI.server.CTTNSessionManagerImpl*. The *-d* option sets the output directory for the generated files. The additional *-v1.2* specifies that clients will be using at least Java version 1.2 with the then used new version of the underlying stub protocol JRMP. Nonetheless, it would be possible to generate compatibility versions for 1.1. With regard to their increased size and the vanishing use of that old versions, it seems justifiable to omit compatibility with versions lower than 1.2.

```
rmic -classpath bin/ -d bin/ -v1.2
de.lmu.ifi.CTTN.RMI.server.CTTNSessionManagerImpl
```

Example TODO: Sample call to rmic for generation of stub class file for specified remote object class.

3.3.6 Registration of the CTTN service

With the completion of the last step and the already performed startups of *rmiregistry* and *rmid*, CTTN's RMI interface can now be activated. The necessary method calls are contained in the *main* method of the class *de.lmu.ifi.CTTN.RMI.CTTNServer*. If the *main* method is invoked without any parameters, the host and port of the *rmiregistry* as well as the name to bind to is obtained from the *CTTNConfig* property *rmiURI*. The settings can alternatively be overridden by just supplying a parameter with a different *URI*. Some other properties are always read from *CTTNConfig*. First of all, *rmidURI* contains the host and port of the *rmid* to use. The contents of the property *socketURI* are used during the initialisation of the socket client connecting with the backend. Finally, *rmiObjectPort* may be used to bind all remote objects initialised during sessions to a specific TCP port.

Similar to the *rmid* startup, *CTTNServer* needs some *System* properties to be set in order to work. The first one is the already known *de.lmu.ifi.CTTN.RMI.policyFile* property, which contains the path to the *ActivationGroup*'s policy file. The properties *de.lmu.ifi.CTTN.codebase* and *de.lmu.ifi.CTTN.clientCodebase* have to point to the server and the client project's binary packages. They are used in the policy file as well as in the *main* method of the *CTTNServer* class. If the codebase URLs point to externally visible locations, the clients are able to download the files from these locations whenever necessary. A complete call to the *CTTNServer* class looks like:

```
java -Dde.lmu.ifi.CTTN.RMI.policyFile=/activationGroup/policy/path
-Dde.lmu.ifi.CTTN.codebase=file:///server/binaries/path
-Dde.lmu.ifi.CTTN.clientCodebase=file:///client/binaries/path
de.lmu.ifi.CTTN.RMI.CTTNServer
```

After the execution of this command, the RMI interface of CTTN is ready to

handle client requests.

3.3.7 Expiration of remote objects

The amount of time RMI remote objects remain valid without a renewal of the lease is determined by the *System* property *java.rmi.dgc.leaseValue*. By default, this value is set to 600,000 milliseconds what corresponds to 10 minutes. After this timeframe, expired remote objects may be subject to garbage collection. This behaviour perfectly satisfies the current requirements of CTTN's RMI interface.

3.3.8 Testing the service

To verify the functionality, a sample RMI client is contained in the client project. The class *de.lmu.ifi.CTTN.exampleClients.RMIClient* starts multiple sessions and executes various commands within those session to provide an overview of the RMI interface's usage. The *URI* of the RMI interface endpoint may either be supplied as first parameter or via the *CTTNConfig* property *rmiURI*.

3.4 CORBA

An alternative to the RMI architecture was initially defined in October 1991 by the Object Management Group (OMG). The Common Object Request Broker Architecture (CORBA) is a standard architecture for distributed object systems. In other words, it basically provides the same service RMI does. However, the most significant difference is, that CORBA has always been designed to be independent from specific operating systems, programming languages and implementations. It has been improved until today's current release version 3.0.3 and is still under active developement. Due to it's for IT dimensions quite high age and extensive support from many major software companies, a lot of programming languages offer very mature support for this standard. Sun Microsystems itself is one of the contributing members of the Object Management Group. It is hence self-evident that the current Java 5.0 Standard Edition's CORBA components mostly comply to the standard. To be exact, they conform to CORBA revision 2.3.1 with some minor limitations and modifications. All

details on this topic can be found in a subsection of Sun's Guide to Java IDL at http://java.sun.com/j2se/1.5.0/docs/guide/idl/compliance.html.

The core component of CORBA is the Object Request Broker (ORB), "which enables objects to transparently make and receive requests and responses in a distributed environment"[6]. The attached Interoperable Naming Service adds the ability to assign names to remote objects. In contrast to RMI, many different free and commercial ORB implementations are available. Anyway, Sun distributes its own implementation of an ORB with the Java SE SDK 5.0. Since its abilities are completely sufficient for CTTN's current requirements, the implementation at hand relies on Sun's standard java classes and tools. Thanks to the open design of CORBA, the ORB's implementation can be substituted by an alternative product later on if required with only minimal changes. The ORB of current choice can be started as background process without prior preparations. The optional parameter *-ORBInitialPort* may be used to bind the orb to a different port than its default being 1049.

```
orbd -ORBInitialPort 1060 &
```

Upon the initial startup of an ORB, it already manages the so-called *RootPOA*. POA means Portable Object Adapter as defined by the CORBA specification. An object adapter in general connects requests using an object reference with the proper implementation. The Portable Object Adapter in particular is meant to ensure portability between different ORB products while defining a set of compulsory capabilities. The initially existing *RootPOA* sets the default policies for all objects. Inherited policies can be overridden by creating a new POA as child of an existing one. Thus the policies of remote objects depend on their assigned parent POA's policies.

Once the *orbd* is running, server applications can register objects with it. The ORB then takes care of mapping client requests to the registered objects. Sun's *orbd* already contains a usable Interoperable Naming Service, so this part is already started with the command above. If a remote object should be findable by clients under a specific name, server applications can bind remote object references to custom names by using this Naming Service.

3.4.1 Defining the interfaces

Similar to RMI, the initial task to complete is the definition of remote interfaces. As already mentioned in the RMI subsection, a portable architecture can't make use of Java object class structures without the definition of appropriate mappings. To achieve an independence from concrete programming languages, the interfaces need to be written in OMG's language for this purpose simply called Interface Definition Language (IDL). The basics of IDL are defined in section 3 of the CORB Architecture and Specification[6]. In general, the syntax of IDL is not too different from the usual Java syntax. Modules in IDL are the equivalent of Java's packages. Each module may contain several interfaces which are implemented by the remote objects. The declarations contained in the modules require some mapping efforts. First of all, the datatypes differ between IDL and Java. The most common mapping rules for types and names are summed up in the appendix B. The standard defined in "OMG IDL to Java Language Mapping" provides a complete reference of all necessary mappings. Sun Java 5.0 implements the versions ptc/00-01-08[8] and its subsequent revision ptc/00-11-03[9]. The standard ptc/00-01-06[7] defines the reverse "Java Language to IDL Mapping" and might also be helpful for modifications of the IDL interfaces.

The following excerpt defines a simplified module *CTTN*, which contains a *CTTNException* and the interfaces *CTTNSessionManager*, *CTTNSession* and *CTTNInterval*. The *raises* statement corresponds to Java's *throws* statement.

```
module CTTN {
    exception CTTNException {
    };
    interface CTTNInterval {
        readonly attribute long id;
        string date () raises (CTTNException);
        string display () raises (CTTNException);
    };
    interface CTTNSession {
        readonly attribute string id;
        string getCTTNVersion ();
        string getProtocolVersion ();
        void end () raises (CTTNException);
```

```
        CTTNInterval parseInterval__string(in string value) \
                raises (CTTNException);
        CTTNInterval parseInterval__string__string(in string from, \
                in string to) raises (CTTNException);
    };
    interface CTTNSessionManager {
        CTTNSession startSession () raises (CTTNException);
    };
};
```

The example shows several different properties of IDL:

- Interfaces may declare object attributes. An optional *readonly* prefix may be added to prevent remote modification of the value.
- The methods defined may declare mandatory parameters and must have a return type. The return types used in the example are *void*, the basic type *string* and the interfaces *CTTNSession* and *CTTNInterval*. If another interface is used as parameter or return type, it has do be declared earlier in the IDL document.
- Parameters must be flagged by one of the modifiers *in*, *inout* or *out*. These modifiers indicate the direction in which the information flows from the view of the server.
- For smooth integration in programming languages that don't support method overloading, IDL prohibits the definition of overloaded methods. However, there are common rules for method name creation when mapping overloaded Java methods to IDL. The mangling appends two underscores to the corresponding Java method name. If the method has parameters, the IDL types are appended, each separated by another two underscores. If a parameter is specified with its fully-qualified name, the leading *:: is stripped and all further occurences of :: are replaced by a single underscore.*

There are some more mapping rules and definitions, but they are omitted here for clarity. If more complex constructions are required, the "OMG IDL to Java Mapping"[8] provides all information in a very succinct way.

Once the IDL definition has been completed, a tool called *idlj* is used to generate the necessary bindings for the Java language. Since the resulting Java classes should reside in the package *de.lmu.ifi.CTTN.CORBA.bindings*, the module name *CTTN* has to be translated to this package's name. This is done by using the *-pkgTranslate* option with the original module name and the target package name. The *-td* option defines an alternative target directory for the resulting classes. To keep the resulting packages small, *idlj* is invoked twice. The first call generates the serverside bindings and writes them to the server project's source directory. The second invokation does the same for the clientside.

```
idlj -fall -td src/server \
-pkgTranslate CTTN de.lmu.ifi.CTTN.CORBA.serverBindings
\
src/shared/de/lmu/ifi/CTTN/CORBA/CTTN.idl
idlj -fclient -td src/client \
-pkgTranslate CTTN de.lmu.ifi.CTTN.CORBA.bindings
\
src/shared/de/lmu/ifi/CTTN/CORBA/CTTN.idl
```

As obvious in the two lines to execute, the *-f* option determines the side for which the bindings are supposed to be. Valid values are *client* and *server* for only one side or *all* for both at once. The resulting files for the server side use an inheritance model, so one of the generated classes has to be extended by the implementation of the respective remote object. The downside of this approach is, that the only slot for class inheritance is used for the CORBA bindings. To eliminate this restriction, the *-f* option also accepts the value serverTIE. In an additional run, bindings for the so-called Tie Model are generated by *idlj*. An implementation class based on this model just has to implement an interface declaring the available methods of the remote object. In consequence, the Tie Model allows the developer to inherit the implementation from a custom class. The negative point of this alternative is that it adds one layer of indirection and hence requires a bit of additional processing power. The implementation at hand uses the direct inheritance model, since differences in the thrown exceptions prevent a direct extension of the socket client classes.

3.4.2 The implementation

Each implementing class has to extend an abstract class that has been created as part of the bindings. The class name thereby starts with the corresponding IDL interface name and is suffixed by *POA*. The translated Java interface can be recognised by its suffix *Operations*. It is already declared to be implemented by the POA class, so it doesn't have to be brought up explicitly in the object's implementation.

Except for the *CTTNSessionManager*, the functionality of each remote CORBA object is backed by a socket client object. This underlying object is stored in a protected object field. Calls that don't return a reference to a new remote object just have to forward the call to the socket client object and return the resulting value. If a method call should create a new remote object and return a reference to it, it has to activate the new object with the ORB to make it available to the client. Currently, all CTTN CORBA objects share the same policies. In consequence, they can also be assigned to the same POA. The current object's POA can be easily retrieved by calling *_poa()*. A following call to the *activate_object_with_id* method of the resulting POA is then used to bind the new object to the same POA as the existing one. Besides the servant object, the method takes an identifier as parameter. This identifier is uniquely generated from the *sessionId*, the object type and the object's identifier. Once the activation is performed, this identifier needs to be reused to obtain an object reference usable by the client. The current object's POA also provides a method for this purpose, namely *create_reference_with_id*. However, only a common CORBA object is returned instead of a remote stub for the wanted class. As part of the client side bindings, the corresponding helper class provides a *narrow* method to ensure the correct type of the object reference. The result of a call to this method finally yields the wanted client-usable reference to the remote object and can thus be returned.

An attentive reader might have noticed earlier, that the *idlj* command for the server side specifies *-fall* instead of the expected *-fserver*. The additionally created client side bindings are generated to make use of an included helper class. The following sample code creates a reference to a remote object that is registered in the ORB. Afterwards, two alternate source blocks are attached that both yield a stub for the remote object. The first version shows the lines of code necessary to solve the task without the helper class of the client side bindings. The second

way illustrates the use of the narrow method that is part of the helper class.

```
// Create reference to object in ORB
Object objRef = this._poa().create_reference_with_id(objName.getBytes(),
    sessionServant._all_interfaces(null, null)[0]);

// a) Necessary code without Helper class
org.omg.CORBA.portable.Delegate delegate =
    ((org.omg.CORBA.portable.ObjectImpl)objRef)._get_delegate ();
de.lmu.ifi.CTTN.CORBA.serverBindings._CTTNSessionStub stub =
    new de.lmu.ifi.CTTN.CORBA.serverBindings._CTTNSessionStub();
stub._set_delegate(delegate);
return stub;

// b) Code with Helper class
return CTTNSessionHelper.narrow(objRef);
```

It is fairly obvious, that the second solution noticeable reduces the amount of code. In consequence, the resulting better clarity of sources using the client side bindings has been favoured over the slightly smaller size of server-only bindings.

3.4.3 Starting the service

Once all object implementations are written, the CORBA interface of CTTN can be started. The necessary commands are contained in the *CTTNServer* class in the server CORBA package. Similar to the RMI version of the file, an *URI* can be used to bind the service to a different ORB. If none is passed upon invocation, the default value is taken from *CTTNConfig*. With this *URI* at hand, a new ORB instance can be initialised. The static *init* method of Sun's ORB expects the host and port settings to be passed as *String* array instead of an *URI*. Since the conversion is required for the clients as well, it has been sourced out to the static *getOrbArgs* method in the shared *CORBA_URIHelper* class. By calling the *resolve_initial_references* method on the new ORB instance, the server application can retrieve CORBA objects for the *RootPOA* and the *NameService*. As already describe above, the *narrow* method of the helper classes is used to obtain stubs for the wanted remote objects. In order to maximise flexibility and to create

an isolated namespace, all CTTN remote objects are bound to the *cttnPOA*. This new POA is currently created as child of the *RootPOA* without overriding any inherited policies. If necessary, policies can thus be adjusted easily. After activating the *POAManager* of the *cttnPOA*, a *CTTNSessionManager* object is instantiated and activated in the *cttnPOA*. The subsequently obtained remote object stub is then bound to the *NameService* with the name that is contained in the ORB's *URI*. A final call to the ORB's *run* method blocks the current thread and waits for invokations from clients.

3.4.4 Expiration of remote objects

3.4.5 Testing the service

The class *de.lmu.ifi.CTTN.exampleClients.CORBAClient* contains a sample client for CTTN's CORBA service. The executed commands resemble those of the exemplary RMI client. Analogously, the *URI* of the CORBA service can either be passed as first parameter upon invocation or be defined by the *corbaURI* property of *CTTNConfig*.

3.4.6 RMI over IIOP — An alternative?

As of the CORBA 2.0 specification, ORBs may intercommunicate using the General Inter-ORB Protocol (GIOP). GIOP itself is defined transport-indepent, so various transport mappings have been defined by the OMG to allow communcation over different network types. The special TCP/IP version of GIOP is called Internet Inter-ORB Protocol (IIOP). All ORBs claiming to support at least the 2.0 revision of CORBA must support this protocol.

Regular RMI implementations don't use IIOP. They use the JRM Protocol instead, so interoperation between RMI and CORBA applications is normally impossible. Sun tries to fill this gap with its implementation of RMI-IIOP, which replaces JRMP with IIOP. The clear advantage of this approach is, that the server application developer doesn't need to explicitly care about the IDL mappings and special implementations in order to make his program usable by CORBA clients. Once the RMI server version is written, the *rmic* tool can also be used to automatically generate IDL files from the existing RMI interfaces. A sample call for

RMI's *CTTNSessionManager* interface would be:

```
rmic -idl -classpath ../client/bin -d
dirForOutput -idlModule de.lmu.ifi.CTTN CTTN
de.lmu.ifi.CTTN.RMI.CTTNSessionManager
```

Although the generation of IDL definitions is much simpler using this command, the thereby resulting files didn't meet the expectations. First of all, the mere number of 29 different IDL files for a single RMI interface seems a bit wasteful. This high number is caused by the fact, that every class inheritance down to the *Object* level is mapped to IDL definitions. The sample IDL definition included ealier in the documentation showed, that 5 lines of code are absolutely sufficient to declare the *startSession* method together with the surrounding interface and module. Besides this rather unbalanced behaviour, the use of RMI-IIOP entails several restrictions for the RMI interface implementation. One of them is the prohibition of the *UnicastRemoteObject* class, which is extensively used in the current RMI server implementation. A complete list of linked limitations is available at http://java.sun.com/j2se/1.5.0/docs/guide/rmi-iiop/rmi_iiop_pg.html#Restrictions.

Due to the unwanted implications on the RMI code and the almost confusing diversity of IDL files, the RMI-IIOP solution has been abandoned. The chosen separate approach requires a bit more maintenance, since the IDL interface has to be created manually. Nonetheless, this additional work ensures that the interface declaration remains compact and that the implementation isn't affected by any restrictions.

3.5 The web service interface

The third way to access the CTTN interface system remotely is via a web service. Among the provided remote interfaces, web services are the most recent technology. They are designed to provide standard means of interoperatibility between software applications, independently from platforms and frameworks they are running on. This is mostly achieved by the exclusive use of XML for all kinds of transferred data. In the meantime, the use of XML is very widespread. In consequence, the necessary parsers are available for most computer systems. A

resulting major advantage is that web services can be used by almost every client target system.

The implementation at hand uses the SOAP protocol in versions 1.1 and 1.2. SOAP itself "is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment"[1]. Additionally, the Web Service Description Language (WSDL) is used to describe the web service. This includes information on defined datatypes, the available methods, ways of transportation and default service endpoints. The API documentation is also integrated in the service's description.

3.5.1 Prerequisites

In contrast to RMI and CORBA, Sun's Java SDK doesn't comprise tools for the creation and consumption of web services in the current release. To fill this gap, Sun offers the Java Web Services Developer Pack free of charge. Nonetheless, many processes aren't yet automated in this product, so the developer has to create a lot of code manually. Since the definition of the SOAP standard in 1998, various alternative projects tried to provide fully-automated frameworks for providing and using SOAP services. However, many of them suffer from poor performance and are therefore abandoned for the use in CTTN's SOAP interface. Another criteria that is part of the goals of this works is the availability free of charge, preferably as open source. Among the available choices, the Axis project fulfiled the selection criteria best and was chosen therefore. It is able to create the WSDL description for a web service automatically and afterwards to create clientside bindings from this very description. Despite the high level of automation offered, Axis is often referred to as one of the fastest, non-commercial implementations available.

Apache Axis is housed at the Apache Software Foundation at http://ws.apache.org/axis/. It has to be installed on an application server for serious use. Right now, Jakarta Tomcat is employed because of a recommendation by the Axis team. It is also available on the Apache Project's websites at http://tomcat.apache.org/. The installation should be done following the tutorials on the respective websites, since they are the most up-to-date source of information. The SOAP interface has been developed and test with Axis 1.3 and Tomcat 5.5.9. Anyway, the use of future minor releases is strongly encouraged since they often fix security-

related issues.

In the meantime, a reimplementation effort of Axis simply called Axis2 is about to reach release quality. In difference to the original Axis, it is based on an improved architecture with extended capabilities. The developers also claim to achieve an improved performance over the previous project. Unfortunately, some of the needed features are still on the experimental feature list and showed several deficiencies during their use. In the current release 0.94, Axis2 still can't be used to replace the original Axis implementation. Nonetheless, some of the new Axis2 features come in very handy for development. Right here, the new deployment model has to be mentioned since it offers so-called "hot deployment". This ability eliminates the need to restart Axis2 in order to use the newly deployed Java archive. Besides that, additional supported standards like WS-Security may be used to further improve the service offered by the CTTN SOAP interface that has been developed as part of this work. Due to these benefits and despite of the handicaps associated with the current releases, an Axis2-based version of the CTTN web service has been developed in parallel. The necessary packages to install Axis2 on the application server are available at the project's homepage located at http://ws.apache.org/axis2/. When the remaining issues are fixed and Axis2 reaches production quality, it is advisable to change to this parallel development tree.

3.5.2 Differences to RMI and CORBA interfaces

The most commonly used pattern of application interaction defined by SOAP is the Remote Procedure Call (RPC). This pattern allows clients to invoke procedures on distant hosts while, in general, waiting for the result. In contrast to RMI and CORBA, SOAP does not define a common representation of object references. In consequence, SOAP is also unable to offer a standard way of method invocation on remote objects. It is up to the web service developer or the utilised framework to create and handle object references. Axis for instance provides support for a session scope, thus isolating the environments of different client sessions. Unfortunately, the way sessions are handled in Axis isn't backed by any standard, so there is no guarantee that other implementations will properly handle the additional session information. When it comes to object references as return value of procedure calls, Axis finally can't offer any direct solution at all. Even if it could, the client side support would anyway be undetermined again.

As already mentioned, one of SOAP's most important features is it's portability — and portability is also one of the primary goals of the CTTN interfaces. For that reason, the current implementation tries to reduce the risks of incompatibility with clients as far as possible. To achieve this goal, proprietary solutions for session handling and object references have to be avoided. The finally chosen way to solve both problems is entirely covered by the SOAP specification and thus has to be supported by every client claiming to be compliant to the standard.

3.5.3 The implementation

The way CTTN clients' procedure calls are processed here is very different from the way RMI and CORBA work. Whenever a client invokes a procedure via SOAP that would normally return a remote object stub in RMI or CORBA, the instance of the underlying socket client class is first written to an according *Hashtable* with a unique identifier as key. Instead of returning a stub, the client just gets an unique identifier for the remote object as result of the procedure call. The identifier object obtained this way doesn't provide any methods that might be really useful to the client. It instead is intended to be passed as parameter to future procedure calls. If it is, the server implementation is able to read the contained identifier information in order to look up the socket client object in the respective *Hashtable*.

A consequence of the different way of referring to remote objects is, that there is no new endpoint associated with every new remote object. Therefore, a single endpoint provides all methods of every available remote object class. The class that contains all these procedures for CTTN's SOAP interface is *de.lmu.ifi.CTTN.SOAP.CTTNImpl*. It also contains the already mentioned *Hashtable*s, that are in fact cascaded on two levels. Since all objects are part of one specific session, they can be grouped accordingly. An inner class called *SessionData* has been created within the *CTTNImpl* class for this purpose. Its *session* object property contains the *CTTNExtendedSessionImpl* socket client object. All other socket client objects are distributed on several *Hashtable* object properties of the *SessionData* object. Each of those *Hashtables* is only responsible for instances of one specific socket client class. Various helper methods take care of adding new socket client objects to them or retrieving the stored objects from them. The *SessionData* objects themselves are created during the start of a new session. They are then added to the static *sessionDatas* property of the

CTTNImpl class.

A cascaded structure working this way requires an additional initial look up to retrieve the respective *SessionData* object. Depending on the total amount of needed session-specific objects for the execution of a single procedure call, the total complexity of object access in the cascade is at worst twice as high as in a plain structure. However, the additional amount of time to access an object in a *Hashtable* is negligible on the presumed server hardware. The advantage of an object grouping by sessions appears whenever a session is terminated by a client request or due to expiration. A plain structure would then require serious filtering efforts over all existent objects to determine, which of them are part of the particular session and thus can be purged. The employed formation instead just needs to remove a single *SessionData* object from the static *Hashtable* to release all session objects for garbage collection at once.

The secure session identifier which is used as key for the *SessionData* objects has already been mentioned in the socket client documentation. Its generation is performed during the creation of a new *CTTNExtendedSessionImpl* instance and can be accessed by the *getId* method. The word "secure" is of special relevance when using SOAP calls. SOAP itself is XML-based and thus passes messages in a human readable format. The use of a simple incremented counter value as session identifier would almost invite potential attackers to modify the contents of surrounding sessions. As described in the socket client documentation, the security of session identifiers is garanted by making them irreproducible. This is especially achieved by the *CTTNConfig* property *sessionSalt* which needs to be set compulsory by every CTTN interface server administrator. Details on the way identifiers of sessions are generated are available in the socket client documentation.

The remote object identifiers that are exchanged between SOAP server and client for referring to remote objects need some explanation. Unlike remote object references in CORBA for instance, they are not of a plain string type. If they would be, client programming languages with the ability to check types couldn't distinguish between different remote object classes. To make this feature work, a separate identifier class for every socket client object class has been created in the *de.lmu.ifi.CTTN.SOAP* package. They are recognisable by the suffix *Id*. Each identifier object has an primary identifier property *id* and a *getId* method that returns its value. All objects except instances of *CTTNSessionId* need another

property *sessionId* for the session identifier to make the identification unique. If present, a *getSessionId* accessor method provides access to this property's value.

This might be a good place to give an example of how client calls via SOAP are processed. Assuming an already created client session, the steps to create a new CTTN interval shall be described here. A call to the procedure to parse an interval requires the *CTTNSessionId* object to be passed as first parameter followed by the interval's string representation as parameter number two. The server then looks up the *SessionData* object which is referred by the value of the passed *CTTNSessionId* object's identifier property. The *getSession* method of the *SessionData* object yields the *CTTNExtendedSessionImpl* object, on which the *parseInterval* method is then invoked. The resulting *CTTNIntervalImpl* socket client object is added to the *intervals Hashtable* of the *SessionData* instance. A new *CTTNIntervalId* object is subsequently constructed with the session's and the new interval's identifier. Afterwards, it is returned to the client as result of the procedure call. Further calls to the *date* procedure for example could then be executed by the client with this *CTTNIntervalId* object as first parameter.

The first attempt to create the *CTTNImpl* class was based on the fact, that overloaded methods are permitted by the SOAP and the WSDL specification. For example, the call to the *date* procedure just mentioned should be directed to *CTTNImpl*'s respective method accepting a *CTTNIntervalId* object. Alternatively, there is another date method which provides the same functionality for CTTN's points. To verify the correct assignment, the *id* property of *CTTNPointId* has temporarily been set to the *int* type to make its property types and names identical to those of *CTTNIntervalId*. Unfortunately, some tests with different client programming languages showed that the assignment to a method is sometimes incorrect under these circumstances. The standard *SOAPClient* provided by PHP for example produced the following output, when calling the *date* procedure with a *CTTNPointId* object as parameter:

```
PHP Fatal error:  Uncaught SoapFault exception: \
    [soapenv:Server.userException] de.lmu.ifi.CTTN.CTTNException: \
    No interval with id 1141084800 in .../CTTN/java/soapTest.php:15
Stack trace:
#0 .../CTTN/java/soapTest.php(15): SoapClient->__call('date', Array)
#1 .../CTTN/java/soapTest.php(15): SoapClient->date(Object(stdClass))
#2 {main}
```

```
thrown in .../CTTN/java/soapTest.php on line 15
```

As visible in the error message, a *CTTNException* occurred because of a non-existing interval with the identifier of the point object. A following analysis of the exchanged SOAP messages showed, that PHP indeed passes the object type of the identifier object with the SOAP call. Nonetheless, Axis seems to be unable to correctly determine the object type of the first parameter. Axis tries to find a procedure with a matching parameter on the property level anyway. Since the *sessionId* is identical for all objects of one session and the *id* property type has been set to *int* for both classes during the tests, Axis simply didn't know which procedure version to select. The error above shows cleary that it picked the wrong one. A subsequent test with a sample Perl client revealed a similar erroneous behaviour. Unfortunately, several different attempts to fix the incorrect assignment on the server side didn't solve the problem. The use of the newest available CVS version also showed no improvements at that point. A pure clientside solution is no option since it would be programming language specific and would thus contradict the goal of maximised portability.

In short, the currently available versions of Axis don't allow the use of overloaded methods in the CTTN interface implementation. To circumvent this issue, all procedure names are prefixed by an object class identifier followed by an underscore. For clarity, this is done for all procedures no matter if they are overloaded or not. The identifier is formed by taking the remote object's simple class name and stripping the leading *CTTN*. In the case of overloaded methods within a single remote object, a suffix consisting of two underscores and a counter value is appended from the second procedure definition on. In the case of the *date* procedure for example, the procedure name with a *CTTNIntervalId* parameter is *Interval_date*. Subsequent overloaded procedures would be called *Interval_date__1*, *Interval_date__2* and so on. This naming convention eliminates the risk of incompatibilities while allowing an comparatively easy, automatic procedure name deduction. The last point is especially relevant for the *UniversalClient* documented later. The *CTTNImpl* class declares all externally visible methods following this naming scheme.

3.5.4 Deployment of the web service

Besides the implementing class, Axis and Axis2 need some meta-information about the web service like its name, the object class name, object class mappings et cetera. In Axis version 1, all these details have to be supplied in a Web Service Deployment Descriptor (WSDD) file. WSDD is yet another XML format, whose base element is either deployment or undeployment. Any number of service subelements may be used to manage services with the specified names. Within the service elements, various options may be specified to modify the service upon deployment. The descriptor currently used for CTTN is contained in the file *cttnDeploy.wsdd* within the *deployment* folder. The following shortened excerpt illustrates its basic structure.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
      xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CTTN" provider="java:RPC">
    <documentation>
      <!-- ... -->
    </documentation>
    <parameter name="className" value="de.lmu.ifi.CTTN.SOAP.CTTNImpl"/>
    <parameter name="allowedMethods" value="*"/>
    <parameter name="scope" value="Application"/>
    <parameter name="wsdlPortType" value="CTTN"/>
    <parameter name="wsdlBinding" value="CTTNSoapBinding"/>
    <parameter name="wsdlServiceElement" value="CTTNService"/>
    <parameter name="wsdlServicePort" value="CTTNServicePort"/>
    <namespace>http://CTTN.ifi.lmu.de</namespace>
    <beanMapping qname="cttn:CTTNSessionId" xmlns:cttn="http://CTTN.ifi.lmu.de"
        languageSpecificType="java:de.lmu.ifi.CTTN.SOAP.CTTNSessionId"/>
    <beanMapping qname="cttn:CTTNIntervalId" xmlns:cttn="http://CTTN.ifi.lmu.de
        languageSpecificType="java:de.lmu.ifi.CTTN.SOAP.CTTNIntervalId"/>
    <beanMapping qname="cttn:CTTNPointId" xmlns:cttn="http://CTTN.ifi.lmu.de"
        languageSpecificType="java:de.lmu.ifi.CTTN.SOAP.CTTNPointId"/>
    <!-- ... -->
    <operation name="session_getCTTNVersion">
      <parameter name="sessionId"/>
      <documentation>
```

```
        <!-- ... -->
      </documentation>
    </operation>
    <!-- ... -->
  </service>
</deployment>
```

The first two levels of the hierarchy inform Axis, that a web service named "CTTN" is about to be deployed. The options on the third level serve the following purposes:

documentation: The first occurrence of documentation describes the service itself. The description will then be present in the resulting WSDL description. It is written in javadoc style to allow an automatic integration in the client's API.

Parameter className: Specifies the name of the class that should be made available as web service.

Parameter allowedMethods: May be used to restrict the public available methods. Since the SOAP class is written exclusively for the web service, a wildcard is used to allow access to all public functions.

Parameter scope: The scope of the web service may be either *Application*, *Request* or *Session*. As mentioned earlier, the way session handling is performed by Axis might reduce the compatibility, so *Session* isn't usable. A *Request* scope would instantiate a new object on every call. Since all necessary information is contained in class properties, the *Application* scope is the best alternative. By its use, only one shared singleton object is created to service all requests.

Parameters wsdlBinding, -PortType, -ServiceElement and -ServicePort: These parameters override the default naming scheme for the different WSDL elements. The chosen names try to make the resulting client side method and class names as short and memorable as possible.

namespace: The default namespace is generated by creating an URI from the reversed package components, so it would be *http://SOAP.CTTN.ifi.lmu.de*. The new setting removes *SOAP* from the namespace.

beanMapping: A beanMapping is one of the available default mappings. It automatically generates the according WSDL types for classes that follow the JavaBean-style get and set accessors. A mapping has to be defined for each object class that is used in parameters or return values. Their respective superclasses also have to be listed with this mapping.

operation: The operation with the specified name and parameters doesn't have to be listed here in order to work. The sole reason for the presence of all operations in the descriptor is to add some documentation to them. Like the service description, the operations' documentations will be included in the automatically generated WSDL description and in the resulting clientside bindings.

A complete list of options is available in the Axis reference guide at http://ws.apache.org/axis/java/reference.html#DeploymentWSDDReference.

Before Tomcat and the contained Axis installation is started, the server archive has to be made available in *axis/WEB-INF/services* in the Tomcat's *webapps* folder.

3.5.5 Starting the service

Once the archive is in place, Tomcat can be started by executing:

```
$CATALINA_HOME/bin/startup.sh
```

Once the server is running, the deployment of the web service to the Axis framework can be executed by using the class *org.apache.axis.client.AdminClient*. For this purpose, its *main* method has to be called with the path to the service's WSDD file as first parameter:

```
java org.apache.axis.client.AdminClient
cttnDeploy.wsdd
```

The service description has now been integrated in the file *WEB-INF/server-config.wsdd* which is located in the axis web application. Without prior changes

to the configuration, the WSDL description of the new web service is available at http://localhost:8080/axis/services/CTTN?wsdl.

One problem of the original Axis is encountered once the underlying classes of the deployed service need to be changed. At first, an update of the used archive has to take place. A subsequent redeployment of the web service will however show no effect. To make Axis use the new archive, the whole application has to be restarted.

The deployment procedure of Axis2-based web services is a bit different. Unlike the original Axis, Axis2 supports so-called hot deployment. This means, that the class archive used by a web service can be imported into a running Axis2 application without a restart. To make the management even easier, the deployment itself is no longer based on an external tool. The only neccessary step to deploy a web service is copying an archive file to the directory *WEB-INF/services* in the Axis2 application folder. The archive file used for CTTN is quite similar to the regular server package with only minor changes. At first, the package extension has to be changed from *jar* to *aar*. In addition, a slightly modified deployment descriptor has to be added to the archive as *META-INF/services.xml*, which is located in the deployment folder in this project. The archive can of course easily spare any RMI and CORBA packages. The current structure of the Axis2 descriptor is quite similar to the previously used one. However, a complete reference of the new format is intentionally omitted here. Since the automatic WSDL generation is still an experimental feature, the descriptor structure might experience some changes until the final release. For that reason, the most up-to-date source of information regarding the format should be used. Unfortunately, the URL of these details currently changes with every release, so interested readers have to find their own way starting at http://ws.apache.org/axis2/.

After copying the created *aar* archive file, is only needs to be copied to the deployment destination. Afterwards, the URL of the new web service will be http://localhost:8080/axis2/services/CTTN?wsdl if the default settings have been used.

3.5.6 Generation of the client side bindings

The next task to perform is the generation of clientside bindings to the webservice. Axis also provides a special class for this purpose with the name

org.apache.axis.wsdl.WSDL2Java. Besides the URL of the web service's WSDL description, several options can be specified as parameters to modify the behaviour of the class' *main* method. Currently, two options are used. One — *-p* — determines the package name of the resulting classes. The second option is *-o*, which sets the output directory to the client project's source folder. The complete call to the *WSDL2Java* tool generating the CTTN client side bindings is:

```
java org.apache.axis.wsdl.WSDL2Java \
-p de.lmu.ifi.CTTN.SOAP.bindings -o src/client \
http://localhost:8080/axis/services/CTTN?wsdl
```

Attempts to generate the bindings from the Axis2-based web service currently fail. The reason therefore seems to be the strange looking and apparently erroneous content of the automatically generated WSDL description. Tests with PHP and Perl clients failed likewise. The current experimental development state of this Axis2 component simply seems to prevent the generation of valid output. Subsequent releases will hopefully solve this problem.

3.5.7 Testing the service

The class *de.lmu.ifi.CTTN.exampleClients.SOAPClient* contains a sample client for the CTTN web service. The matter of executed commands is identical to the one of the exemplary RMI and CORBA clients. The *URI* of the web service is obtained from the *soapURI* property of *CTTNConfig*. Alternatively, this value can be overriden by passing an URI as first parameter upon invocation.

4 Clientside design and implementation

As mentioned in the description of the general structure, the required client side bindings have already been generated as part of the server implementation. So once the common prerequisites including the inclusion of the client side jar archive, a client is able to use the CTTN system via the available RMI, SOAP or web service interface.

Which architecture is finally chosen is completly up to the user's preference. It should be anyway considered that the offered remote interfaces differ in their performance. Experience during deveolpment showed, that RMI should be the best solution for any client running Java. Alternatively, CTTN's CORBA implementation reaches almost the same performance while still making the objects directly available to the client. The web service is primarly intended to be a compatility option for clients that are unable to use the other interfaces.

If the use of another interface than RMI is intended, it should be considered to use the *UniversalClient* instead of the native bindings. If the interface has to be changed later for any reason, client software just needs to use an alternative *URI* instead of adapting all CTTN calls to another interface's different API.

4.1 The RMI interface

For use with RMI, the client may also skip the inclusion of the *cttnClient.jar* file in the *CLASSPATH*. Alternatively, the involved classes can be obtained from a remote repository. To enable this feature, the client has to include an according *URL* in the *System* property *java.rmi.server.codebase*.

Before methods on a remote server can be invoked, the client needs to set a *SecurityManager* that allows communication with the distant host. Besides that, the sample CTTN RMI clients are able to use *CTTNConfig* to obtain the remote server's address. In consequence, they need access to the filesystem to read the *config.properties* file. Sun's Java implementation already provides a *RMISecurityManager* class that defines a set of methods to check the permission for different types of actions.

```
System.setSecurityManager (new RMISecurityManager() {
```

```
    public void checkConnect (String host, int port) {}
    public void checkConnect (String host, int port, Object context) {}
    public void checkRead(java.io.FileDescriptor fd) {}
    public void checkRead(String file) {}
    public void checkRead(String file, Object context) {}
  });
```

This specific *RMISecurityManager* in fact permits the client software to connect to any distant host and to read every file the executing user has access to. For a productively used client application, this definition is of course way to wide and should be modified to allow only actions that are strictly required. During the development of a client, this is however a adequate definition since it is simple and prevents exceptions caused by erroneously set too narrow permissions.

Once the *SecurityManager* is set, a client has to obtain a so-called stub for a remote object from the server. A stub object transparently hides communcation with the server while implementing the interface of the remote object. The interface implemented by the initial remote object of CTTN is *de.lmu.ifi.CTTN.RMI.CTTNSessionManager*. The look-up of the remote object stub is achieved by using the RMI name service. The according call to the static *lookup* method of *java.rmi.Naming* has to be performed with a string representation of the server's URI string.

```
CTTNSessionManager sessionManager =
(CTTNSessionManager) Naming.lookup("rmi://server.to.use/NameOfCTTN")
```

;

If this call succeeds, the remote CTTN system is yet ready for use. The only difference to the invocation of local objects' methods is that every call to the remote CTTN system may additionally throw a *java.rmi.RemoteException*. This exception indicates errors that result of broken network connections and alike and hence needs to be caught by the client application.

The class *de.lmu.ifi.CTTN.exampleClients.RMIClient* contains an examplary client that uses RMI to communicate with the CTTN system. It conducts some simple operations and prints out status information. The *main* method accepts the URI of the remote object as first parameter. If it isn't set, the value of the *rmiURI* property is attempted to be read via *CTTNConfig*.

4.2 The CORBA interface

As far as exception handling is concerned, CTTN's CORBA interface is a little bit different from the other interfaces. Instead of using the *CTTNException* from the *de.lmu.ifi.CTTN* package, a CORBA-specific version out of the corresponding subpackage is used. It extends *org.omg.CORBA.UserException* and is declared in the interfaces IDL definition. Besides this one, clients also have to catch any instance of *SystemException* from the same package. This type of exception if thrown upon errors that are related to the system of remote object access and method invokation.

The method names partially differ from the ones used in RMI. Since CORBA abandons the use of overloaded methods, affected method names have to carry a suffix. The way this appendix is create is described in the server side documentation in detail. For pure client use, this knowledge is not strictly required.

The steps to obtain the initial remote *CTTNSessionManager* object are a bit more complex than their RMI counterpart. The following example uses an *URI* to determine the remote endpoint to use and the name of the remote object in the Naming Service.

```
// Initialise the ORB on the specified address
ORB orb = ORB.init(CORBA_URIHelper.getOrbArgs(uri), null);
// Get the root naming context
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
// Look up the remote object reference in naming (without the trailing slash)
String remoteName = uri.getPath().substring(1);
CTTNSessionManager sessionManager = CTTNSessionManagerHelper.narrow(ncRef.reso
```

After performing these actions, the CTTNSessionManager is ready to be used for starting sessions.

An exemplary client for this remote interface is available in the *de.lmu.ifi.CTTN.exampleClients* package. The class called *CORBAClient* optionally accepts the URI of the remote service to use. If none is provided, the property *corbaURI* is read from *CTTNConfig* for this purpose. If the interaction with the remote system succeeds, the main method will

start some sessions, perform some simple remote calls while printing out details on the tasks.

4.3 The SOAP interface

The third available way to communicate with a CTTN interface server is via SOAP. A corresponding web service is currently based on Apache Axis For client side use, some jar archives that are part of the Axis distribution currently have to be included in the client's *CLASSPATH*. Which archives are required in concrete is documented on the project's website.

Due to limitations of the current SOAP specifications, remote objects cannot be used directly by remote applications. In CTTN's SOAP interface implementation, referring to remote objects is therefore accomplished by the use of identification objects. In consequence, all SOAP procedures are offered by a single endpoint. Whenever remote object need to be referenced, the idenfitication objects are simply passed as parameters when calling a procedure.

To start using the SOAP interface of CTTN, the following lines of code have to be included in the client application:

```
CTTNServiceLocator cttnService = new CTTNServiceLocator();
CTTN cttn = cttnService.getCTTNServicePort(uri.toURL());
```

The resulting CTTN object is then ready for use. The supplied *URL* may be omitted. In that case, the default URL contained in the WSDL description is used as endpoint.

Due to compatibility issues, method overloading had to be abandoned in the current implementation. The first consequence of this fact is that every method name is preceeded by the abbreviated class name of the remote target object followed by an underscore. Names of overloaded methods are also suffixed by two underscores and a unique integer value to ensure their uniqueness.

Similar
to the other available interfaces, the package *de.lmu.ifi.CTTN.exampleClients* contains an exemplary client class called *SOAPClient*. Like the other samples, it executes various CTTN operations in multiple sessions. The endpoint's URI may be supplied either as first argument or via the *soapURI* property of CTTNConfig.

4.4 The UniversalClient

The direct use of the interface implementations described in the previously is definitely the most storage efficient approach. Additionally, it involves the lowest possible amount of overhead when calling a method on a remote object. The disadvantage connected with the direct use is the dependency on the selected transport protocol. CORBA's own type of exceptions in Java and completely different use of the SOAP interface make a seamless migration between different remote interfaces impossible.

Especially when processing power and storage space aren't crucial, the lacking exchangability of remote interface protocols used in a client isn't desirable. If the availability of server interfaces isn't determined in advance, this lack might even compromise the usability of the CTTN system in custom projects. To address this issue, it is necessary to introduce an abstraction layer hiding the underlying implementation. The main requirement of such a layer is the implementation of common interfaces for all communication types.

CTTN's clientside realisation of this abstraction layer is called *UniversalClient*. The package is located in CTTN's base Java package. The implemented interfaces are the same as used by the RMI client implementation. The RMI interfaces were chosen because of their immediate definition in Java.

The obvious way to implement the outlined abstraction is to wrap all classes in newly created classes that implement the RMI client's interfaces. In consequence, every single method would have to be defined to forward the call to the real communication object with its specific interface. In addition to this task, every exception in every affected method declaration would have to be mapped from the interface-specific class to its RMI counterpart. The definition of a new exception type might yet lead to the necessity of touching every single method mapping again. The creation of the subclasses would consequently be a very time-consuming mission. Besides these downsides, every additional method of the CTTN core system would cause an in-line growth of the resulting UniversalClient classes.

An alternative is the use of Java's built-in reflection API introduced with version 1.3. By the use of reflection, information regarding the structure of objects can be obtained at runtime and even methods can be intercepted and invoked dynamically. The difference between the last approach and this one is, that the

definition of the mapping from the remote interface method signatures to the ones provided by the underlying communication classes is done in a more abstract way. The advantage achieved thereof is that the reflective implementation doesn't need to be modified when a remote interface changes. A change would only be necessary if the abstract mapping rules are violated by the new interface. The downside of an abstract definition is that the Java compiler doesn't complain about nonexistent method calls at compile time. Hence, the detection of errors is deferred to runtime. To overcome this limitation, it is strongly encouraged to define and run appropriate *junit* test cases when CTTN's interfaces reach a final state. With these precautions in mind, the outlined reflective solution seems to be the favourable way to map the method calls.

For implementation, the reflection class with the needed features is *java.lang.reflect.Proxy*. Its purpose is the interception of method calls on interfaces specified at runtime. To create a new *Proxy* instance, the static method *newProxyInstance* is used. The arguments of the method are the *ClassLoader* to use, an array of interface classes to implement and the *InvocationHandler*. The interface *InvocationHandler* located in the reflection package declares an *invoke* method. Any calls to the proxy interfaces will be dispatched to the implementation of this method.

The mapping and calling part of method interception is partially sourced out to the *CTTNMethodMapper* class family. The abstract class *CTTNMethodMapper* itself contains the static *getMethodMapper* method. It returns an instance of the mapper subclass assigned to the provided class. Additionally, an abstract method *callMethod* with some parameters containing details on the original method call and the remote objects involved is declared. The subclass implementations of this method are responsible for the modification of calls and the execution of the resulting calls. All method mapper implementations cause the throw of a *NoSuchMethodException* if they are unable to find a matching function on the target object of their choice. The currently defined subclasses are:

***CTTNCORBAMethodMapper*:** This mapper incorporates CORBA-specific details of method name mapping. However, the first action performed is an attempt to call the original method on the target object. This is done for performance reasons since many methods don't need to be altered. Under some circumstances, this first attempt will fail. As described in the CORBA subsection earlier in this document, the

tool *idlj* generates bindings with a different naming convention for getters and setters. For that reason, a JavaBean style *get* and *set* prefix needs to be stripped and the first character of the remaining string has to be converted to lowercase. Since CORBA doesn't support overloading, it might subsequently be necessary to search for a method with an altered name. The naming convention defined by the OMG is too complex to be fully implemented in this class. However, the amount of parameters can be extracted from the name of the overloaded method with reasonable efforts and is therefore checked. In consequence, this method mapper examines all methods until it find's one with an according base method name, a suffix indicating the right amount of arguments and finally arguments of the correct number and class types.

***CTTNSOAPMethodMapper*:** Calls backed by the SOAP interface require the most complex modifications. Similar to the CORBA variant, this SOAP-specific mapper tries to call a method identical to the one called on the proxy object on the target object first. This is done since the SOAP binding classes provide some methods themselves. If this attempt fails, the call must be diverted to the remote object. As described in the SOAP subsection above, there is currently no way to directly address remote objects via SOAP. That's why calls to remote SOAP procedures pass an object identifier as first parameter. The only exception to this rule is the *CTTNSessionManager* since it is used in a static way. The next major difference compared to other mappers is the target of the call. All offered procedures are available via a single SOAP endpoint. For that reason, the target of the method call is altered to the *initialTarget* object which contains the required initial endpoint. Finally, method overloading is unusable due to difficulties that are explained in the documentation of the server implementation. For that reason, each method name is prefixed by the abbreviated simple object class name of its corresponding object and a following underscore. Additionally, method names can be suffixed by two underscores plus a unique number. An according search for method following this naming scheme is performed in this last step.

***CTTNDefaultMethodMapper*:** This subclass is the straightest member of the mapper class. It simply forwards the call from the proxy object to the target object. One possible target object set for this mapper is the

socket client. The implemented interfaces are almost identical to their RMI counterparts, the only difference is that no *RemoteException* is declared. The other possible target group is the RMI client itself. Anyway, it must be emphasized here that wrapping them is quite futile since they already implement the wanted interfaces. The only reasonable cause to wrap them might be for debugging or logging purposes.

Once a method wrapper returns a result and it's a local object, there's nothing else to do except returning it. But whenever the return value of the call is another remote object, the underlying call returns an object of a class that doesn't implement the needed RMI client interface. For that reason, *CTTNProxy*'s *invoke* method checks if the returned object is of the desired type. If it is not, it simply wraps it in a new instance of the *CTTNProxy* class with the needed interface and the returned target object. In consequence, calls to that object will run through the same steps just described whenever one of it's methods is invoked.

For all remote protocols, several different steps are necessary to obtain a reference to the remote object from the server. Since the target of this additional layer is to hide all kind of differences from the client, the class *CTTNUniversalClient* defines two overloaded static methods *getSessionManager*. The first version takes an *URI* as parameter specifying the protocol and the endpoint of the remote object. Version number two can be called without any parameters. It is identical to the first version, but the remote object's *URI* is read from the *defaultURI* property, which is hopefully contained in the *CTTNConfig* configuration. If it is not, a *CTTNException* is thrown. Depending on the protocol chosen, the methods look up the remote object while mapping all protocol-specific exceptions to their RMI complements. The return value will be the native remote object in the case of RMI, under all other circumstances it will be a CTTNProxy instance implementing the RMI interface *CTTNSession* and wrapping the real remote object. Two other static methods called *startSession* accomodate the fact, that most clients won't ever need to start more than one session. For that reason, they internally call the *getSessionManager* methods and return the result of a subsequent call to the *startSession* method of the obtained object. In the rare case of multiple session within one client, calling *startSession* on the object implementing *CTTNSessionManager* should anyway be preferred because of the lower communication overhead.

5 Further development of the interfaces

Whenever changes to the interfaces themselves are necessary, various tasks have to be performed. To simplify development, all relevant steps can be handed over to the Apache Ant tool. It is a Java-centric replacement for build tools like *make* or *gnumake*. The definition files are based on XML to support different platforms at once. The contained shell commands for CTTN however are exclusively targeted on linux systems. The software is available under the Apache Software License Version 2.0 at http://ant.apache.org/ in its current 1.6.5 version.

Currently, a single Ant file called *build.xml* provides all necessary tasks of the entire project. Its first lines defined various properties containing mostly paths and URLs as well as some *CLASSPATH*s. The defined tasks care about compilation, API and package creation as well as possibly necessary cleanups of the various paths. They also take over the generation of server and client side bindings.

If the provided interfaces need to be altered, the changes need be performed from the core system to the remote interface client. In consequence, the first affected part of the interface project is the socket client, followed by the server implementation and finally the client side bindings. The order of compilation and bindings generation in the ant file is harmonised with this process. The required server classes for instance are generated before the generation of bindings and the compilation of client files takes place.

The currently implemented functional range of the interface server has been intentionally reduced to provide only some basic objects with only a limited amount of methods. This decision has been made since the core server is still under development. Several of its abilities are either under development or not yet available for use over the socket protocol.

6 Summary

List of Figures

References

[1] Nilo Mitra. Soap version 1.2 part 0: Primer. W3c working draft, World Wide Web Consortium (W3C), 2003. See http://www.w3.org/TR/soap12/.

[2] Hans Jüergen Ohlbach. Computational treatement of temporal notions — the CTTN system. In François Fages, editor, *Proceedings of PPSWR 2005*, Lecture Notes in Computer Science, pages 137–150, 2005. See http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2005-29.

[3] Hans Jürgen Ohlbach. Fuzzy time intervals and relations – the FuTIRe library. Technical Report PMS-2004-4, Inst. f. Informatik, LMU München, 2004. See http://www.pms.informatik.uni-muenchen.de/mitarbeiter/ohlbach/systems/FuTIRe.

[4] Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-28, Inst. f. Informatik, LFE PMS, University of Munich, June 2005. See http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-28.

[5] Hans Jürgen Ohlbach. Modelling periodic temporal notions by labelled partitionings of the real numbers – the PartLib library. Research Report PMS-FB-2005-27, Inst. f. Informatik, LFE PMS, University of Munich, June 2005. See http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-27.

[6] Object Management Group (OMG). The common object request broker: Architecture and specification. Technical Report formal/99-10-07, Object Management Group (OMG), October 1999. See http://www.omg.org/cgi-bin/doc?formal/99-10-07.

[7] Object Management Group (OMG). Java language to idl mapping. Technical Report ptc/00-01-06, Object Management Group (OMG), January 2000. See http://www.omg.org/cgi-bin/doc?ptc/00-01-06.

[8] Object Management Group (OMG). Omg idl to java language mapping. Technical Report ptc/00-01-08, Object Management Group (OMG), January 2000. See http://www.omg.org/cgi-bin/doc?ptc/00-01-08.

[9] Object Management Group (OMG). Omg idl to java language mapping. Technical Report ptc/00-11-03, Object Management Group (OMG), November 2000. See http://www.omg.org/cgi-bin/doc?ptc/00-11-03.

Appendix

A *CTTNConfig* properties

The available options are:

socketURI: The host and port of the URI specified here will be used for connection with the CTTN backend server. Example: telnet://localhost:1953/

sessionSalt: The string supplied by this option is used as a prefix when generating the session's id. To garant the isolation of sessions, the value should be set to a random string consisting of at least 8 characters including letters, digits and special characters. Example: Z2379!ASDzh...

rmiURI: The host and Port

B CORBA mapping rules

The following mappings affect Java code creation from IDL definitions. Alternatively, another overview of the IDL to Java mapping is available on Sun's website at http://java.sun.com/j2se/1.5.0/docs/guide/idl/mapping/jidlMapping.html. For a complete reference see the "OMG IDL to Java Language Mapping" as defined in ptc/00-01-08[8] and its subsequent revision ptc/00-11-03[9] as well as ptc/00-01-06[7] defining the "Java Language to IDL Mapping". Details on IDL itself are available in section 3 of the CORBA specification[6].

IDL type	Java type
boolean	boolean
[w]char	char
octet	byte
[w]string	java.lang.String
[unsigned] short	short
[unsigned] long	int
[unsigned] long long	long
float	float
double	double
fixed	java.math.BigDecimal

Table 1: Basic Mapping of IDL to Java types

Java language keywords					
abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	this	throw
throws	transient	try	short	static	super
switch	synchronized	void	volatile	while	
Additional Java constants					
true	false	null			
Methods of java.lang.Object					
clone	equals	finalize	getClass	hashCode	notify
notifyAll	toString	wait			

Table 2: Reserved method names that will be prefixed with an underscore.