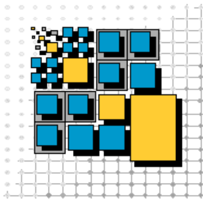

Otto-Friedrich-Universität Bamberg

Distributed and Mobile Systems Group

Manual
OCoN Framework version 1.0

Jens Bruhn, jens.bruhn@wiai.uni-bamberg.de
Sven Kaffille, sven.kaffille@wiai.uni-bamberg.de

Lehrstuhl für Praktische Informatik
Otto-Friedrich-Universität Bamberg
Feldkirchenstr. 21
D-96052 Bamberg
GERMANY

16.06.2004

Contents

1 Introduction **1**

2 Concepts **2**

2.1 Scheduling . . . 2

2.2 Internal Administration . . . 3

3 Installation **7**

3.1 Requirements . . . 7

3.2 Installation process . . . 7

4 Configuration **8**

4.1 Preparation . . . 8

4.2 Middleware infrastructure . . . 8

4.3 OCoN Framework startup . . . 10

4.3.1 Starter . . . 10

4.3.2 Responsible creator . . . 11

4.3.3 Configuration . . . 12

5 Programmer's Guide **15**

5.1 Service Implementation . . . 15

5.1.1 Defining the interface of a service . . . 15

5.1.2 Providing a service implementation . . . 16

5.2 Client Implementation . . . 20

5.3 Logging . . . 24

5.4 Compiling the framework . . . 24

List of Figures

1 Protocol net of `ResetableWarehouseSection` 3
2 Example situation. 3
3 Framework Architecture 4
4 Relations between framework classes and service implementation. 17
5 Relations between framework classes and client implementation. 21

1 Introduction

The OCoN Framework is an implementation of a hierarchical scheduling framework for state-based services which is published under GNU General Public License (GPL)[1]. This document contains all information needed to run a system based on the framework. Additionally developers will find information on how to implement services and clients which make use of the framework.

The remainder of this manual is structured as follows. First section 2 gives an overview over the basic concepts of the framework. These are divided into two parts. The first part deals with concepts which belong to the scheduling itself. The second part discusses the internal administration of a system. This section lays the foundation for the explanations of the latter ones. In section 3 the installation process is described. This includes the requirements for the framework's installation as well as a summary of the installation results. The 4. section explains how the framework can be used to establish a running system. This includes the necessary preparations, the startup of the middleware infrastructure and the instantiation of framework components. In this context all relevant configurations as well as all needed files are discussed. The last section of this document contains the programmer's guide. First the concepts which are relevant for service and client implementations are highlighted. The next two parts describe the implementation of services and clients. A short description of logging follows. The section ends up with compilation instructions for the framework.

[1]The GPL Version 2 can be obtained from [FSF91].

2 Concepts

2.1 Scheduling

The framework provides an easy way to implement state-based services. Methods of such a service can only be executed in certain states. To provide the methods a service implements an interface. Additionally to the pure interface description in Java [2] a description of the relationships between states and invokable methods of services is needed. Such a description is provided – according to the OCoN aproach[3] – via so called `protocol nets`. For this implementation the protocol nets are specified within an XML[4] file. Such a file – from now on called `protocol-net-file` – contains the possible states of a service and the transitions between these states through the use of methods defined by its interface. If the interface of a given service is part of an interface hierarchy, this information is also part of the protocol-net-file[5]. According to the protocol-net-file services are organized within a hierarchy by the framework so that common methods of more specialized services can be used instead of methods of the super interface. Consequently, specialized services can act as more general ones. To achieve this, services connect to so called `scheduler`s. There exists exactly one responsible scheduler for each service type (each service interface) in the whole system. It holds references to currently running instances of services. Additionally to the interface and the protocol net a service can be associated with auxiliary information. This information is provided by a so called `template`. Clients can request service types for invocation of methods from the scheduler for the particular service type. One service instance is then assigned to the client for method invocation. This assignment depends on the template a client desires the service to have and on the current state a service is in. If there is currently no service matching the template the client desires, the client is notified about that. If there is currently no service in a state that allows the client to make its invocation the client has to wait until the service transits to a state that allows the invocation.

Example: The running example for this manual is a warehouse where different warehouse sections are available as services. Items can be supplied to these sections or consumed from them. Only one item can be supplied to or consumed from a particular section at any time. The type of this service is called `WarehouseSection`. There is also a specialized type of a warehouse section from where all items it contains can be consumed at once. This type of section is called `ResetableWarehouseSection` for the remainder of this manual.

In figure 1 the representation of the `ResetableWarehouseSection` protocol net is shown. The shadowed transitions for supply and consume indicate that these methods are inherited from `WarehouseSection`.
For these types of example services we assume the following situation. There are three `WarehouseSections` (W) and two `ResetableWarehouseSections` (RW) available. The framework organizes these instances as shown in figure 2.

If a client wants to use a service, it requests execution of a service method from the

[2] Java runtime environments are available at [SM04a],
[3] Additional information can be found at [WGG97].
[4] See [BPSM+04]
[5] Currently only one direct super interface is supported by the framework for each interface. Consequently multiple inheritance is not supported.

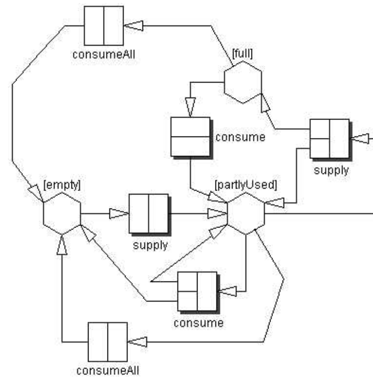

Figure 1: Protocol net of `ResetableWarehouseSection`

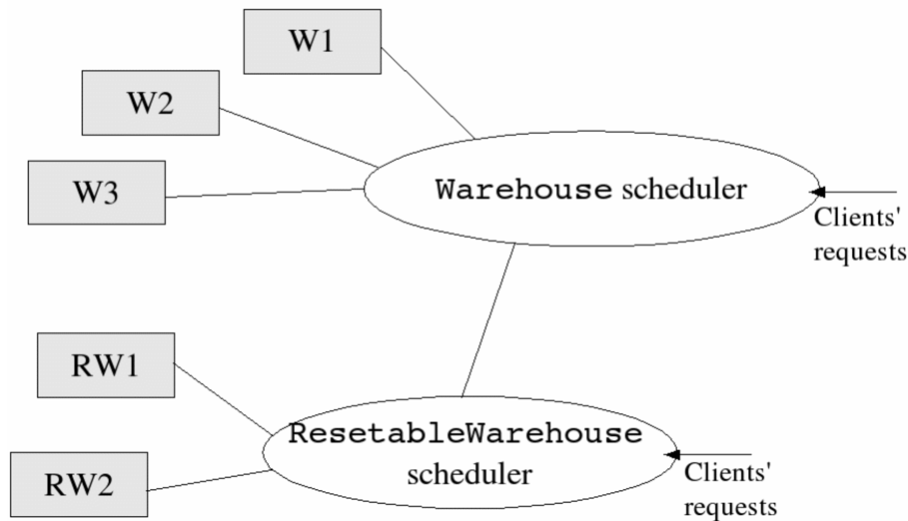

Figure 2: Example situation.

scheduler that is responsible for the desired service type. The scheduler assigns a service that is in a state that allows execution of the method requested by the client. In our example it is transparent to clients that request services of type `WarehouseSection` which instance from this example situation is assigned to them by the scheduler [6]. If an instance of `ResetableWarehouseSection` is assigned to such a client it can use this like a `WarehouseSection`. Therefore instances `W1` through `RW2` (if in a state matching the request of a client) can be assigned to a client that requests a service of type `WarehouseSection`. Requests from clients which desire a `ResetableWarehouseSection` can only be assigned to the instances `RW1` and `RW2`. Services can provide templates as mentioned above to facilitate the request of a specific instance the and clients can search for services with a template they desire a service to have.

2.2 Internal Administration

The architecture of the framework is divided into three perspectives called `views` in the remainder of this document. Figure 3 shows these views separated by the vertical dashed lines. The `Service view` contains all parts of the framework necessary for communication

[6]Only services whose states match a client requests can be assigned.

with the other views from the services' point of view. The `Client view` can be seen as counterpart for the client side. The `Internal view` groups all components needed for the internals of the framework and for the interaction with parts of the other views.

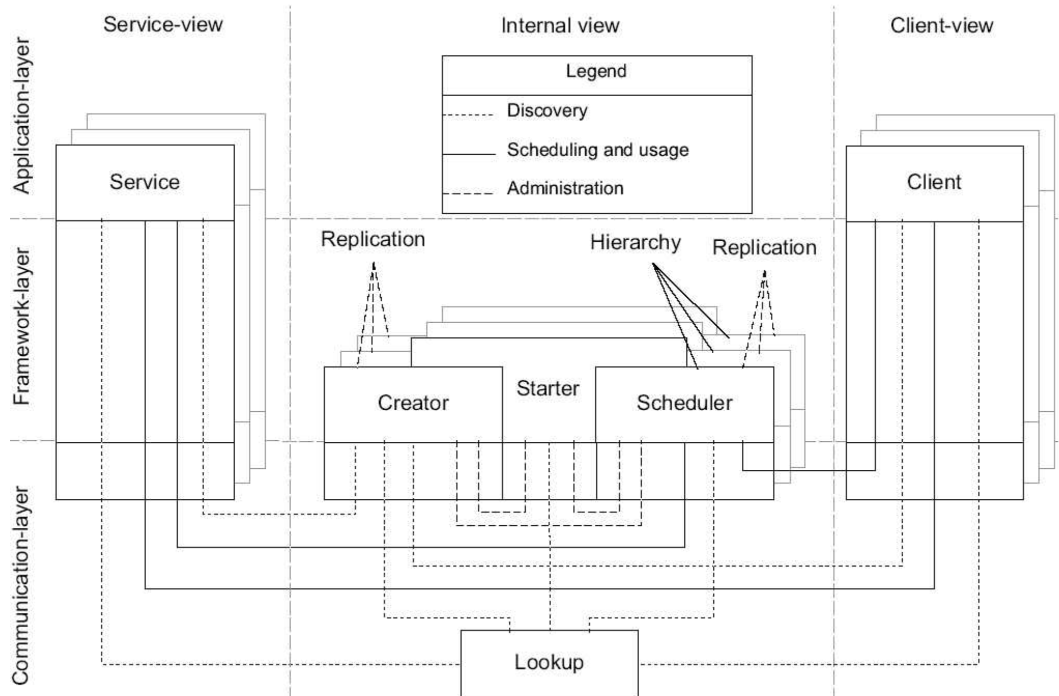

Figure 3: Framework Architecture

The layers of the architecture are separated by horizontal dashed lines and can be considered to be orthogonal to the views. The `Communication layer` represents the communication infrastructure of the framework. It is responsible for the delivery of messages between the participants of a system based on the framework. To enable participants to find an initial entry point to the system, this layer also provides a so called `Lookup`. For the upper layers of the framework this layer provides interfaces for interaction and abstracts from the concrete communication infrastructure. This encapsulation allows the migration to another middleware as long as the interfaces are still provided. Consequently, a migration would only affect the `Communication layer`.
Above the Communication layer the `Framework layer` resides. It contains all parts of the framework that are not directly visible for developers of clients and/or services. Namely, these are the `Starter`, `Creator` and `Scheduler` components. The components are started within a system and take over different tasks like administration and scheduling. Starters are the platforms for the other two components. Upon these new creators and/or schedulers will be instantiated during runtime if needed. The main creator is responsible to initiate the creation of a new replicated creators or schedulers. At any time there is only one main creator allowed to exist. Other creator instances act as replicas which are able to take over responsibility in case the responsible one is shut down or has crashed. Schedulers are the core components of the framework. They deliver the scheduling facility for clients and services as described above. Schedulers are replicated similar to the creators.
The highest layer is the so called `Application layer`. It provides an API for developers of clients and/or services. It hides the internal steps of scheduling and allows developers of clients to develop them as if they interact with the desired services directly. There does not exist an `Application layer` for the internal view, because developers are not intended to manipulate the internal behavior of the framework directly.

Three different groups can be identified for the communication between the participants of a running system . `Discovery` (dotted connections in figure 3) of other components of a system is done in two steps via the lookup and the main creator. First a component – which desires to establish a connection to another component – requests a reference to the responsible creator via the lookup. The creator keeps track of all internal components of the framework. References to these components (e.g. starters) can be obtained from the creator.
Every direct and indirect (via schedulers) communication between services and clients and among schedulers is subsumed within the `Scheduling and usage` group of communication links that are highlighted in figure 3 via continuous connections.
Dashed connections represent communication links between creators and starters, creators and schedulers and vice versa for internal `Administration`.

In order to start up a system based on the framework at least one `lookup` has to be started first. In a second step the first `starter` has to be instantiated. The responsible creator is started by this `starter` . Every subsequent starter which is instantiated requests a reference to the responsible creator from the lookup and registers with it. The main creator keeps track of all starters within the system and uses them to instantiate new creators or schedulers.
If a service wants to be accessible by clients and is not yet connected to a correspondent scheduler, it first requests a reference to the main creator via the lookup. This reference is used to request a reference to the scheduler which is responsible for the particular interface. If such a scheduler already exists – because of other services already connected to it – the creator hands back a reference to the scheduler. Otherwise a new scheduler for the interface is instantiated upon a starter. This is only possible if the instantiation would not exceed the limitations of the starter[7]. If the limitations of all starters within the system are reached, the request will result in an exception. If a new scheduler is instantiated, schedulers for all interfaces which lie above in the interface hierarchy of the desired interface are also started up. This is done because clients which want to use a more general interface should be able to get a reference to a correspondent scheduler.
If a client wants to use a service of a particular interface and it does not hold a reference to a correspondent scheduler, it first requests a reference to the main creator from the lookup. From the main creator it requests a reference to a scheduler serving the desired interface. If such a scheduler already exists – because a least one service for the interface is already known within the system – the desired reference is handed to the client. Otherwise nothing is returned. In contrast to the proceeding for a service, no new scheduler will be instantiated. This would only make sense if a service is known for the particular interface or a more specific interface. In this case a correspondent scheduler would already exist.
A scheduler gets `idle` if there is no service or lower scheduler left to serve. The existence of clients does not matter because their requests cannot be fulfilled anymore. In this situation the scheduler as well as all its replicas will be shut down after a certain period of time. This proceeding guarantees that a long running system does not grow infinitely and allows it to react to newly appearing services for new interfaces.

Within a running system creators and schedulers are replicated. The number of instantiated replicas depends on three aspects. An upper bound for replicas of creators and schedulers for each interface can be specified[8]. Only if there exist enough starters this

[7]See section 4.3.3
[8]See section 4.3.3

upper bound can be reached, because replicas are only instantiated by different starters. Thus, no two creator replicas or replicas for one and the same scheduler are started on the same starter. Even if there exist enough `starters`, their limitations can lead to a minor number of concurrently running replicas than it was desired[9].
The responsible creator propagates information about newly discovered or created framework components (starters, creators, and schedulers) as well as the crash or shutdown of them to its replicas. With this information a creator which takes over responsibility (e.g. after the currently responsible creator has crashed) has all knowledge it needs to fulfill its duties.
Schedulers replicate component information (if exist) about clients, service and lower schedulers which are connected to them, about the higher scheduler they are connected to, and about known replicas. Moreover, information about new and fulfilled requests is also replicated. This information is sufficient for a scheduler which takes over responsibility for a particular interface. The system has a built-in self healing facility (implemented by the creator) which provides a high degree of fault tolerance and a minimized need for human administration during runtime. Three situations are conceivable when intervention by an administrator is required:

1. All lookup facilities have crashed. This leads to a situation where the components are not able to find each other anymore[10] and system checks made by the components result in exceptions. In this situation an administrator has to startup at least one lookup.

2. All creators have crashed or have been shut down and no replicas are left. In this situation no new components will be instantiated and new services or clients cannot participate in the system. This can be considered as similar to the previous point, but the cause is different. In this situation an administrator has to startup one creator.

3. If the limitations of all starters are reached, no new schedulers can be instantiated even if this is desired by services for interfaces for which no schedulers exist. This situation can only occur because of two possible reasons or a combination of them:

   - The capacities of the starters were not sufficiently dimensioned. Otherwise this situation cannot occur.
   - Starter(s) have been shut down or have crashed without compensation.

   For this situation there exist two possible solutions which both lead to an enhancement of capacity. An administrator can startup additional starters or reconfigure existing starters to allow them to instantiate more framework components[11].

[9] See section 4.3.3

[10] The system would work further on for components which are already interconnected, but new services and clients will not be able to participate. Moreover, crashes and/or shutdowns cannot be handled.

[11] **Attention**: Starters which were reconfigured have to be restarted. If this is not done carefully this can lead to the second situation.

3 Installation

3.1 Requirements

The installation of the framework is performed with help of a Java-based installer packaged in an executable jar file. In order to run this installer there must be a Java version 1.4 compatible runtime environment installed on the system. In order to execute the framework the dedicated system has to meet the following software requirements.

- An operating system for which a Java Virtual Machine is available.
- An installation of a Java 1.4 compatible Software Development Kit.
- An installation of Jini Technology Starter Kit v. 2.0_001[12].

3.2 Installation process

Execute the jar file to install the framework . In the directory where the installer is located, type at the command line type:

```
java -jar ocon_framework_installer.jar
```

The graphical installation user interface appears. Follow the steps given by the installation wizard. You should install the byte code or source code[13]. After the installation process has finished successfully you should find (depending on the installation options you have chosen) the following subdirectories under the `ocon_framework` installation directory.

- `bin`: This directory contains scripts to start the framework's components.
- `classes`: This directory contains the byte code.
- `config`: This is the directory for standard configuration files.
- `docs`: Here this manual and the API documentation can be found.
- `lib`: The files `jini-core.jar`, `jini-ext.jar`, `sun-util.jar` and `tools.jar` have to be placed here for the framework to work. If the logging should be used also a jar file named `log4j.jar` and containing at least `log4j` version 1.2.8[14] must be placed here.
- `src`: This directory contains the complete source code of the framework.

[12] Jini can be downloaded from [SM04b].
[13] If you only install the source code see section 5.4 to compile the framework by yourself
[14] Log4j can be obtained from [Fou04].

4 Configuration

This section describes the necessary steps to startup a system based on the OCoN Framework. The first subsection explains which preparations are necessary before the actual configuration can take place. The current implementation of the framework is based on `Jini` 2.0_001. For this reason the second subsection describes how to prepare the Jini infrastructure. The following subsection 4.3 shows the steps to startup the system upon the established `Jini` infrastructure. All scripts which are mentioned in this section reside in the `bin` subdirectory of the installation directory if not stated differently. The same holds for all configuration files in the subdirectory `config`. Additionally, scripts are named without their extension. In this case the extensions `.bat` for Windows or `.sh` for Unix have to be added for the particular environment. Otherwise the ending and the correspondent operating system will be mentioned explicitly. Note that some line breaks in listings were inserted for formatting purposes.

4.1 Preparation

In order to use the framework it has to be installed according to the instructions in section 3. Additionally, the necessary scripts to startup the middleware infrastructure and framework components must be generated. This can be done using the script `create-scripts`. The absolute path to the framework installation directory[15] must be provided as first command line parameter. The second parameter must be the absolute path to the Jini installation directory[16]. Based on the pathes the script generates all necessary startup-scripts in the subdirectory `bin` of the installation directory. The particular script only generates scripts for the operating system it is executed on. Additionally, the files in the subdirectory `config`, which depend on the host computer the framework is installed on, are generated.

4.2 Middleware infrastructure

The central component of the infrastructure is the `Jini` yellowpages service, the so called `Reggie`. It is used to provide a reference to the responsible creator of the system. In contrast to other implementations based on `Jini` the OCoN framework does not use `Reggie` to provide references to all services in a system but only to the central responsible component. To use the component properly a `codebase` for the service has to be started up. It allows clients to obtain classes which are used in the context of the interaction with the yellowpages service. For the current implementation it is based on an HTTP server that comes with the `Jini` packages. The server can be started by using the `StartHTTP` script. Listing 1 represents the basic structure of the script for Unix. The relevant parts of the file for Windows based systems (`StartHTTP.bat`) are quite similar, so that separate listings for both operating systems are left out here. This holds for all following listings if there are no significant differences for the correspondent operating system.

[15] The path used here should not end with a slash and must not lead to a subdirectory of the directory, to which the framework has been installed.

[16] Again, no slash should appear in the end of the path and the path must not lead to a subdirectory like `lib`.

Listing 1: Script StartHTTP.sh for Unix

```
export INSTALL_DIR=/<jini-basedirectory>
java -jar $INSTALL_DIR/lib/tools.jar -port 8081 -dir
<jini-bsaedirectory>/lib -verbose -trees
```

Inside the script the server is instructed to start listening at port 8081 for the standard configuration generated by the `createscripts` script. If this is not desired, the particular part of the generated script can be adjusted. If this is done ensure that the script in listing 2 as well as the file in listing 4 are changed in a consistent way. Listing 2 represents the script to start the `Jini` yellowpages service.

Listing 2: Script StartReggie.sh for Unix

```
export INSTALL_DIR=<jini-basedirectory>
export CONFIG_DIR=../config
java -cp ../classes -Djava.security.policy=$CONFIG_DIR/policy.all
-Djava.rmi.server.hostname=<creation-hostname>
-Djava.rmi.server.codebase=http://<creation-hostname>:8081/
-jar $INSTALL_DIR/lib/start.jar
$CONFIG_DIR/start-transient-jrmp-reggie.config
```

It can be seen from the script that the service is instructed to direct clients to the previously started codebase if they need to download `.class` files. Additionally, the location of a so called `policy file` is specified. This file is needed for security purpose. In the standard configuration this file allows everything as it can be seen in listing 3. For further details about policy files refer to the correspondent website at [SM02].

Listing 3: Standard policy file for Reggie

```
grant{
    permission java.security.AllPermission;
};
```

The reader is strongly encouraged to adjust this file according to his/her needs to achieve a higher degree of security. The third information handed to the `Reggie` service is the location of a configuration file. The structure of this file can be seen in listing 4.

Listing 4: File start-transient-jrmp-reggie.config

```
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.start.ServiceDescriptor;
 com.sun.jini.start {
  private static codebase = "http://<creation-host>:8081
        /reggie-dl.jar";
  private static policy = "<framework-basedirectory>/config
        /policy.all";
  private static classpath = "<jini-basedirectory>
        /lib/reggie.jar";
  private static config = "<framework-basedirectory>/config
        /transient-jrmp-reggie.config";
  static serviceDescriptors = new ServiceDescriptor[] {
   new NonActivatableServiceDescriptor(codebase, policy, classpath,
     "com.sun.jini.reggie.TransientRegistrarImpl",
     new String[] { config }
   )
  };
}
```

Inside this file again information like codebase and policy is provided. This file is not discussed in further detail because changes other than the locations of the codebase or the policy file are beyond the scope of this manual. Please refer to the `Jini` documentation for further details.

After executing the script `StartReggie` the yellowpages service should be up and usable as middleware foundation for the system. It is possible to start multiple `Reggie`s with one codebase for all of them or multiple codebases without any conflicts. Moreover through such a proceeding a higher level of fault tolerance can be achieved on the middleware layer. On the other hand a higher degree of network load would be the consequence.

4.3 OCoN Framework startup

Before starting the framework infrastructure it has to be ensured that the `Jini` infrastructure has been started properly. For the given implementation this means that the yellowpages service in combination with the correspondent codebase must be started in advance. There are two different kinds of scripts that start framework components. These are described in the following subsections. The first subsection deals with the startup of a so called `Starter` and the second one describes the startup of the initial responsible creator. In the context of the second one the basic concept of creators for the establishment of a leading creator is described shortly. The main configuration parameters as well as their influence on the system behavior are explained in section 4.3.3.

4.3.1 Starter

The basic component of the framework infrastructure is the `Starter` which acts as instantiation platform for creators and schedulers. A script for the startup of a `Starter` called `startplatform` is generated during the preparation process described in section 4.1. The structure of this file is shown in listing 5.

Listing 5: Script startplatform.sh for Unix

```
export CLASSPATH=$CLASSPATH:../lib/jini-core.jar:
    ../lib/jini-ext.jar:../lib/sun-util.jar:../lib/tools.jar:
    ../lib/log4j.jar:../classes
java -cp $CLASSPATH -Djava.rmi.server.hostname=<hostname>
    -Djava.rmi.server.codebase=http://<hostname>:8082/
    -Djava.security.policy=../config/policy.all
    de.uniba.wiai.lspi.ocon.framework.main.PlatformStarter
```

The first part of the script sets the needed classpath in addition to the already existing classpath. In this path the `log4j.jar` file is contained. This is only needed if `log4j` is used for logging. Please refer to section 5.3 for further information.

Before the starter can work properly an HTTP codebase has to be started up. This is done in the same way as described in section 4.2 for the codebase of the yellowpages. The script to start the codebase is also generated and named `StartStarterHTTP`. This file is nearly the same as in listing 1 and therefore not listed again here. According to the listing for the startup of a `Starter` the codebase is listening at port 8082. If this should

be changed it must be done in both files `startplatform` and `StartStarterHTTP` in a consistent way.

Again – as in section 4.2 – the standard policy file from listing 3 is used which permits full access. This file should be adjusted according to the particular needs of the execution environment.

Within a system multiple `Starter`s should be instantiated on different computers to gain the advantage of fault tolerance which is part of the framework design. Even the startup of more than one `Starter` upon a single host is possible. This can be done by executing the `startplatform.sh` script multiple times. By doing this all `Starter`s use the same codebase. If each starter should have its own codebase the script has to be copied and adjusted to use another codebase. Additionally, for each `Starter` a separate `StartStarterHTTP` script must exist. It is also possible to use the same codebase for all `Starter`s within a system. This can be achieved by starting the codebase upon a dedicated server and adjusting the startup scripts for the `Starter`s to instruct them to use the common codebase.

4.3.2 Responsible creator

In order to start framework services like `Creator`s and `Scheduler`s at least one `Creator` has to be started manually by a system administrator. This is done by executing a separate script called `startleader` which is shown in listing 6.

Listing 6: Script startleader.sh for Unix

```
export CLASSPATH=$CLASSPATH:../lib/jini-core.jar:../lib/jini-ext.
  jar:
   ../lib/sun-util.jar:../lib/tools.jar:../lib/log4j.jar:../
      classes
java -cp $CLASSPATH -Djava.rmi.server.hostname=<hostname>
    -Djava.rmi.server.codebase=http://<hostname>:8082/
    -Djava.security.policy=../config/policy.all
    de.uniba.wiai.lspi.ocon.framework.main.LeaderStarter
```

The script is similar to listing 5 in section 4.3.1. Internally it starts up a `Starter` and instructs it to instantiate a `Creator`. The proceeding for the instantiation of the `Starter` is done similar to the one described in section 4.3.1. Consequently it is not explained here again. The new `Creator` will search for a responsible one within the system. If it does not find the main creator it will declare itself as leading creator and from that time one will act as administration service for the system. Due to the self regulation ability of the framework, there is no problem if multiple creators are instantiated by the administrator. Moreover, it is possible to startup `Creator`s upon dedicated `Starter`s. To achieve this the `Creator`s have to be started before any `Starter` like described in section 4.3.1 is brought up. This is necessary because otherwise `Creator`s will be instantiated upon them by the leading creator. This is no problem, but can lead to two undesired side effects:

1. It is possible that the total number of `Creator`s will exceed the desired number of`Creator`s within a system. This will not be regulated by the responsible `Creator`.
2. In case of the responsible `Creator` crashes or is shutdown, another one can take over responsibility which resides within a computer which is not dedicated for `Creator`s.

The second effect can happen if the number of actually available `Creator`s falls below a lower bound and in consequence new `Creator`s are instantiated. With the current implementation it is not possible to restrict the creation of `Creator` to special workstations. the only workaround is to set the number of desired `Creator`s to one and startup new ones in case of crashes. Note that the self healing property is not given in such a high degree as for the standard proceeding if this is done.

4.3.3 Configuration

The configuration of the system behavior is done within the `standard.xml` file of which a standard case is shown in listing 7.

Listing 7: Sample framework configuration file

```
<parameters>
 <starter>
  <sleeptime Value="10000"/>
  <maxServiceCount Value="3"/>
 </starter>
 <creator>
  <nets Value="<path-to-net-file>/<net-file-name>"/>
  <min>
   <creators Value="2"/>
   <schedulers Value="2"/>
  </min>
  <max>
   <creators Value="3"/>
   <schedulers Value="3"/>
  </max>
  <critical>
   <creators Value="2"/>
   <schedulers Value="2"/>
  </critical>
  <sleeptime Value="1000">
   <checkinterval Value="2">
    <complete Value="2"/>
   </checkinterval>
  </sleeptime>
 </creator>
 <system>
  <property>
   <communicationFactoryProperty
   Value="de.uniba.wiai.lspi.ocon.framework.com
   .CommunicationFactory">
    <value Value="de.uniba.wiai.lspi.ocon.framework.jini
    .JiniCommunicationFactory"/>
   </communicationFactoryProperty>
  </property>
 </system>
</parameters>
```

This file is generated by the `createscripts` file during the preparations described in section 4.1 and placed into the `config` subdirectory of the installation directory.

The file is divided into three main parts

1. The *<starter>* part contains all parameters regarding a particular `Starter`.
2. The *<creator>* part contains all parameters regarding a particular `Creator` in case one is started upon the correspondent `Starter`.
3. Within the *<system>* part environment variables are specified which should be set at startup of a `Starter`.

These parts will be explained in the following paragraphs.

<starter> The `<starter>` part of the configuration file contains two parameters which have to be specified.

The first parameter is `sleeptime` which specifies the timespan in milliseconds between two system checks which are performed by the correspondent `Starter`. Every `sleeptime` milliseconds the `Starter` will check if the responsible `Creator` is still reachable. If this is not the case, a reference to the new one is looked up. Additionally, all services which were instantiated by the current `Starter` will be checked to ensure that they still work properly. If a service is identified that is not reachable any more the reference to this service will be removed and the service will be logged off at the responsible `Creator`.

With the second parameter `maxServiceCount` the maximal number of active services at the same time for the `Starter` is specified. There is no difference made between `Creator`s and `Scheduler`s because the maximal number of `Creator`s is always one. If the parameter is e.g. set to 3 as in the standard case there will be at most one `Creator` and two `Scheduler`s or zero `Creator`s and three `Scheduler`s running at the same time upon the correspondent `Starter`. This limit can be used to specify the capacity of services for each `Starter`. In which way the different capacities are used by the responsible `Creator` can't be specified by parameters.

<creator> For `Creator`s there are three groups of parameters to specify.

The first group only consists of the parameter `nets`. With this parameter the absolute path to the file which includes the protocol nets is specified. For the discussion of this file refer to section 5.1.1. Note that this file must include all known interfaces. The use of multiple protocol net files is not supported in the current implementation.

The second group includes the parameters subgroups `max`, `min` and `critical`. Within each of these two parameters – `creator` and `scheduler` – exist. In the actual implementation only the subgroup `min` has influence on the behavior of the correspondent `Creator`. It indicates for the framework services `Creator` and `Scheduler` the minimal number of instances which should be active within the system. For `Scheduler`s it is interpreted for each interface separately. If the number of active instances falls below this value, e.g. because of shutdown or crash of a `Starter`, the responsible `Creator` tries to start up new instances to reach the lower bound fixed in this subgroup. For `Creator`s this is only possible if there are enough `Starter`s left upon which no `Creator` is already running and which have not yet reached their limit of services to start that was specified by the parameter `maxServiceCount`. For `Scheduler`s this is nearly the same except that

Starters can be chosen which did not start Schedulers for the same interface as the new one.

The third group consists of the parameter sleeptime. This parameter has the same meaning as sleeptime in the <starter> part. It indicates the time interval in milliseconds between two checks for necessary actions. The encapsulated parameter checkinterval indicates the number of intervals between two system checks. If this parameter e.g. is set to 2 and the parameter sleeptime is set to 10000, every 20 seconds a system check will occur. The encapsulated parameter complete of checkinterval indicates the interval between two deep system checks. If e.g. the parameter is set to 2, every second system check will be a deep system check. The proceeding of system checks is different for a responsible Creator and replicated ones. For a replicated Creator there is no difference between a normal and a deep system check. It will check if the responsible one is still available. If this is not the case an election of a new main creator will take place. During a normal system check the responsible Creator first analyzes, if there is another responsible Creator within the system. This can only occur in exceptional cases, e.g. if network partitions merge. If recognized, the problem will is solved by shutting down all but one main creator. The rest of the normal check deals with possible violations of lower bounds for Creators and Schedulers. If a violation is noticed, it is solved by instantiation of new services up to the desired bound, if possible. During a deep check additionally all references to framework services and Starters are checked to find links to crashed components. If a component is not reachable any more, the correspondent reference is removed. After this, the violation checking of the normal system check is performed. Consequently the deep system checks will causes – especially within a huge system – a great amount of network traffic. So the parameters of the <starter> part should be set carefully and adjusted if necessary. Notice that the settings at the first responsible creator are most relevant because these will be propagated to replicas.

<system> Within the <system> part of the configuration file environment parameters can be specified. These are set at first during startup of a Starter. In the sample file above only the parameter
de.uniba.wiai.lspi.ocon.framework.com.CommunicationFactory
is set to
de.uniba.wiai.lspi.ocon.framework.jini.JiniCommunicationFactory.
This specifies the communication layer factory. Consequently if another middleware should be used only this parameter has to be adjusted.

5 Programmer's Guide

This section describes how to implement services and clients with help of the framework. First the steps necessary to implement a service are described via the implementation of an example service. Based on that the following section describes how a client can use services.

5.1 Service Implementation

This section deals with service implementation and first describes the general proceeding to implement a service. At the end of this section service implementation is shown on the basis of the warehouse example.

5.1.1 Defining the interface of a service

In order to implement a service first the protocol net has to be designed. From this protocol net design the methods are taken and specified in a Java interface. This interface has to extend the framework interface `de.uniba.wiai.lspi.ocon.framework.com.service.-ServiceInterface`[17] or another interface that itself extends `ServiceInterface`. Every method defined by the interface has to throw a `...framework.com.exception.-NetworkException`. [18]
Example: Listing 8 shows the interface of a `ResetableWarehouseSection`.

Listing 8: ResetableWarehouseSection

```
package de.uniba.wiai.lspi.ocon.framework.example.service.
    warehouse;

import de.uniba.wiai.lspi.ocon.framework.com.exception.
    NetworkException;

public interface ResetableWarehouseSection extends
    WarehouseSection{
     public Item[] consumeAll() throws NetworkException;
}
```

The methods `consume` and `supply` are defined by the interface `WarehouseSection`.
The next step is to create the protocol-net-file. Listing 9 displays a cutout from the protocol-net-file for `ResetableWarehouseSection`.

Listing 9: ResetableWarehouseSection protocol-net-file.

```
<?xml version="1.0" encoding="UTF-8"?>
<nets>
<net Interface="...framework.example.service.warehouse.
    WarehouseSection">
     <states>
```

[17] In the following the package name is shortened by replacing 'de.uniba.wiai.lspi.ocon.' with '...'.
[18] Currently this is neither ensured to compile time nor to runtime. The programmer has to take care of this. Defining a method without this Exception could lead to unexpected behavior at a client implementation.

```
      <!-- states here as below-->
    </states>
    <methods>
      <!-- methods here as below-->
    </methods>
    <subnets>
      <net
        Interface
        ="...framework.example.service.warehouse.
          ResetableWarehouseSection">
        <states>
          <state Name="full" ID="1"/>
          <state Name="partlyUsed" ID="2"/>
          <state Name="empty" ID="3"/>
        </states>
        <methods>
          <method Name="supply">
            <prestates>
              <stateref IDREF="2"/>
              <stateref IDREF="3"/>
            </prestates>
            <poststates>
              <stateref IDREF="1"/>
              <stateref IDREF="2"/>
            </poststates>
          </method>
          <!-- other methods here -->
        </methods>
      </net>
    </subnets>
  </net>
</nets>
```

In the first part the description of the super interface `WarehouseSection` is given. It looks similar to the description for `ResetableWarehouseSection` in the second part (for this reason the state and method definitions are left out). In the method section of each interface description every method has to be associated with the states a service can be in before execution and after execution of that method. Listing 9 shows only definition of method supply and the other methods are left out to save space.

5.1.2 Providing a service implementation

In order to provide an implementation for a service interface the class `...framework.service.OCoNService` has to be extended by the implementation. This class provides the basic functionality for a service to connect to the framework, e.g., its scheduler, and manipulation of its template. The service implementation also has to implement the service interface for which the implementation provides the functionality. Figure 4 gives an example of the inheritance relations between a service and the framework classes. The constructor has to call the super constructor `OCoNService(String initialState, String interfaceFile, Class _interface)`. The first argument is the initial state the service is in. States are represented as Strings. The second parameter is the path to the protocol-net-file and the last parameter is the class of the interface that the service

implements. The constructor sets up the connection to the framework and creates a service template as well.

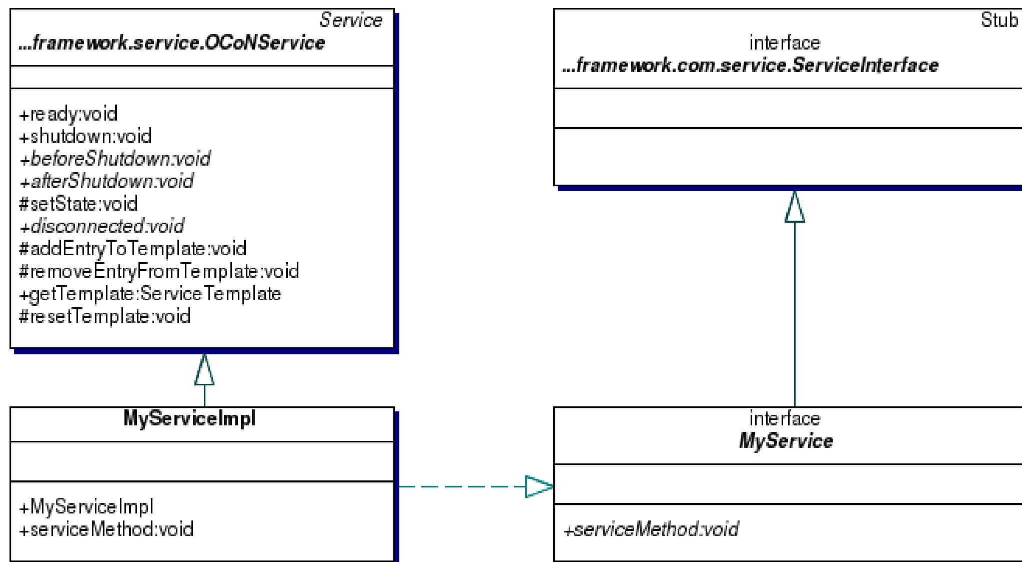

Figure 4: Relations between framework classes and service implementation.

A template consists of several entries that have a key and a value. The key (also called name) is a `String` and a value can be any `Serializable` object. Entries of a template are matched at a service scheduler against the templates that clients desire a service to have for a method invocation. A template T1 matches another one T2 if T1 has a subset of the keys of T2 and the corresponding values of these keys are equal. So it is recommended that the implementations of values overwrite the `equals(Object obj)` method.
To manipulate the template of a service the methods `addEntryToTemplate(String name, Serializable value)`, `removeEntryFromTemplate(String name)` and `resetTemplate()` of class `OCoNService` has to be used. These methods ensure that local changes of the template are propagated to the scheduler. The first one adds the entry with provided name and value to a service template. If an entry with an equal name exists, it is overwritten. The second method removes an entry from the service template and the last one resets the template to an empty one. Templates have a unique identifier, the so-called template id. All template classes and interfaces can be found in package `...framework.template`. For example the interfaces `TemplateID`, which defines the methods of a unique identifier for a template, and `ServiceTemplate`, that defines methods a service template provides, can be found in the package.

To ensure that the assignment of clients to a service works correctly a service implementation has to maintain the transitions from one state to another. This has to be done in each method when the state has changed. In order to set a new state of a service the method `setState(TemplateID templateID, Stringstate)` has to be used. The `templateID` used here can be obtained from a service template by invoking the method `getTemplate()` and on the resulting template the method `getID()`. The `Strings` representing the states should be defined as constants in a service interface to ensure a consistent use of the states. A connection of a service to the framework may break down. In this case the `disconnected()` method is called to indicate this. It is left to the service implementer to decide what to do in this case. For example the administrator of the service can be

notified.

To start up a service an instance of it has to be created. To indicate to the scheduler of the service that the service is ready for method executions the `ready()` method has to be called. This method can be invoked in the constructor or, if some additional setup has to be done, from within the class that created the service.

In order to disconnect a service from the framework the method `shutdown()` has to be called. If some tasks have to be performed before or after a service disconnects this can be done in methods `beforeShutdown()` and `afterShutdown()` which in that case have to be implemented by subclasses of `OCoNService`.

Example: Listing 11 shows an implementation of `WarehouseSection`. The methods `consume()` and `supply()` are defined by the implemented interface `WarehouseSection` which extends `...framework.com.service.ServiceInterface`. The implementation class extends `OCoNService` to be able to connect to the framework.

Listing 10: `WarehouseSection` interface.

```
package de.uniba.wiai.lspi.ocon.framework.example.service.
    warehouse;

import de.uniba.wiai.lspi.ocon.framework.com.exception.
    NetworkException;
import de.uniba.wiai.lspi.ocon.framework.com.service.
    ServiceInterface;

public interface WarehouseSection extends ServiceInterface{

  public final String FULL = "full";
  public final String PARTLYUSED = "partlyUsed";
  public final String EMPTY = "empty";
  public final String SECTION_NAME_ATTRIBUTE_NAME = "name";
  public final String ITEM_TYPE_ATTRIBUTE_NAME = "item";

  public void supply(Item item) throws NetworkException;
  public Item consume() throws NetworkException;

}
```

Listing 11: An Implementation of `WarehouseSection`.

```
package de.uniba.wiai.lspi.ocon.framework.example.service.
    warehouse;

public class WarehouseSectionImpl extends de.uniba.wiai.lspi.ocon
    .framework.service.OCoNService implements WarehouseSection {

  protected final int MAX_ITEMS = 10;
  protected Item[] items = new Item[MAX_ITEMS];
  protected int current = -1;
  protected String current_state = WarehouseSection.EMPTY;

  /** Creates a new instance of WarehouseSectionImpl */
  public WarehouseSectionImpl(String initialState,
                              String interfaceFile,
```

```
                              Class _interface,
                              String name,
                              String item_type) {

      /* Call super constructor to establish connection
       * to framework. */
      super(initialState, interfaceFile, _interface);
      /* Set template information */
      addEntryToTemplate(WarehouseSection.
         SECTION_NAME_ATTRIBUTE_NAME, name);
      addEntryToTemplate(WarehouseSection.
         ITEM_TYPE_ATTRIBUTE_NAME, item_type);
      ready();
  }

  protected String determineCurrentState(){
    if (current == -1) {
        this.current_state = WarehouseSection.EMPTY;
    }
    else if (current == (MAX_ITEMS-1)) {
        this.current_state = WarehouseSection.FULL;
    }
    else{
        this.current_state = WarehouseSection.PARTLYUSED;
    }
    return this.current_state;
  }

  public Item consume() {
    Item currentItem = this.items[current];
    this.items[current] = null;
    current--;
    setState(this.getTemplate().getID(), this.
       determineCurrentState());
    return currentItem;
  }

  public void supply(Item item) {
    current++;
    this.items[current] = item;
    setState(this.getTemplate().getID(), this.
       determineCurrentState());
  }

  public void disconnected() {}
  public void beforeShutdown() {}
  public void afterShutdown() {}

}
```

The constructor `WarehouseSectionImpl` takes the same parameters as the constructor of `OCoNService` and with these calls `super()`. This initializes the connection to the framework. The last two parameters are used as values for the service template. If the template is set up within the constructor it is indicated to the framework that the service is ready to execute methods by invoking `ready()`. The state is maintained in every method

implemented from interface `WarehouseSection` with help of `determineState()`. Before the methods return the state for the service template is set. The last three methods, whose functions are described above, have nothing to do in the example service.

To start the example service, a class with a `main()` method has to be created. This method for example takes as command line arguments the parameters `interfaceFile`, `name` and `item_type`[19] that are required for the `WarehouseSectionImpl` constructor. After creation of the service we have to ensure that the Java Virtual Machine (JVM) it is running in does not shutdown.

Listing 12: Starting the example service.

```
public static void main(String[] args){
  /* check command line arguments */
  ...
  /* start service */
  WarehouseSection section
    = new WarehouseSectionImpl(WarehouseSection.EMPTY, args[0],
                               WarehouseSection.class, args[1],
                               args[2]);
  /* prevent JVM from shutting down. */
  while (true){
    try {
      Thread.sleep(60*1000);
    }
    catch (InterruptedException e){}
  }
}
```

In this example the service runs until the JVM is killed. If you implement services, you are encouraged to make a clean shutdown by invoking the `shutdown()` method. Not invoking `shutdown()` has the same effects as a crash of a service and it takes a certain time for the framework to recognize this.

For a service to work correctly two system properties have to be set. The first one `java.security.policy` must provide the path to a valid security policy file. The second system property `de.uniba.wiai.lspi.ocon.framework.jini.service.codebase.dir` must specify the path to the directory where the classes of the service are located to enable download of these classes for clients.

5.2 Client Implementation

This section describes how a client that wants to use services can be implemented. All classes relevant for client implementation can be found in package `...framework.client`.

A client has to implement the interface `Client` that defines the `disconnectedFrom(String interface_name)` method to notify a client when the connection to the scheduler for interface `interface_name` has been lost. Furthermore a client has to have a reference to an instance of `RequestManager`. Such a reference can be created by a client with the constructor `RequestManager(Client client)` by passing a reference to itself to this

[19] If the example files have been installed the `item_type` parameter can be `TestItem1` or `TestItem2`.

constructor. Figure 5 shows the relations between a client implementation `MyClient` and the classes as described above.

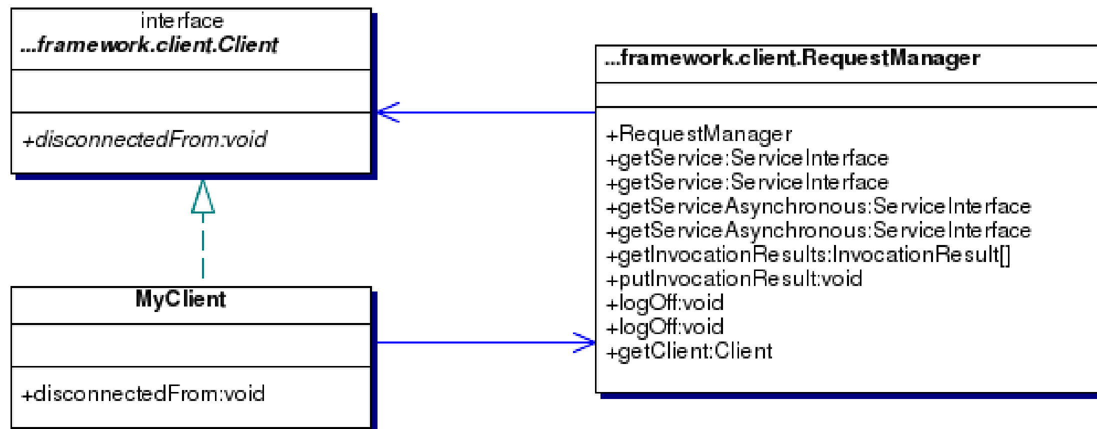

Figure 5: Relations between framework classes and client implementation.

The `RequestManager` is responsible for fetching references to services that the client wants to use. These references (from now on called service proxies) canbe associated with a template. The class to define a template is `...framework.template.OCoNClientTemplate`. This template can be filled with `OCoNTemplateEntry` instances applying the method `addEntry(TemplateEntry entry)`. An `OCoNTemplateEntry` consists of a key/name and a value. These have to be passed as arguments to the constructor `OCoNTemplateEntry(Stringname, Serializable value)`. Templates are compared at the scheduler for arequested service with the available services' templates.

There exist two types of service proxies: synchronous and asynchronous ones. To obtain a reference to a synchronous service proxy a client invokes one of the `getService()` methods of its `RequestManager`. One of these takes a singleparameter of type `Class`, that specifies the interface of the service aclient wants to use. To request a service with a certain template the second`getService()` method can be used which in addition to a `Class`parameter requires as second parameter an instance of `OCoNClientTemplate`.The proxy then is associated with that template. A service proxy provides the interface of the requested service and can be used for subsequent method invocations. If the service proxy is associated with a template, this templateis used for each invocation. Invocations made on synchronous service proxiesblock a client until the invocation has been finished. Asynchronous service proxies can be obtained by invoking one of the `getServiceAsynchronous()` methods, whose paramter structure is the same as for the `getService()` methods. Method invocations on asynchronous service proxies return immediately. If the invoked method has a return value, `null` is returned. The results of asynchronous invocations on a certain asynchronous service proxy can later be obtained from the `RequestManager` with help of method `getInvocationResults(ServiceInterface proxy)`, to which the service proxy (the results are for) has to be provided as a parameter. The return value of this method is an array of `InvocationResults`. This array may have the length 0 if the results are not available yet (or no invocations have been made). From an `InvocationResult` the invoked method can be obtained by `getMethod()` that returns a `java.lang.reflect.Method` object. If the method had a return value this value can be obtained by calling `getResult()` which returns an `Object` that has to be casted into the correct type. If an exception occurred during method invocation this can be obtained by

invoking `getError()`. The return value of this method is a `Throwable` object or `null` if no exception occurred.

Service methods throw at least a `...framework.com.exception.NetworkException`. If this exception is of type `...framework.com.exception.ImpossibleRequestException` this means that a client requested a service with a template that does not exist or if the client requested a service without a template there are currently no services available.

Example: Listing 13 shows the implementation of a client that supplies an item to a service of type `WarehouseSection`. In its constructor it creates a reference to a `RequestManager`. In method `supplyItem()` the client obtains a reference to a service from the previouslyobtained reference to the `RequestManager`. In this method first a template is constructed and then the service is requested. The service can be used to supply items of type `TestItem1` as specified inthe template. Therefore an item of this type is created and supplied to theservice. The `supply()` method throws a `NetworkException` which has to be caught.

Listing 13: A client using a synchronous service proxy.

```
import ...template.*;
import ...framework.client.*;
import ...example.service.warehouse.*;
import ...com.exception.NetworkException;

public class Supplier implements Client{

  protected RequestManager requestManager;

  public Supplier(){
    this.requestManager = new RequestManager(this);
  }

  public supplyItem(){
    OCoNClientTemplate template
      = new OCoNClientTemplate();
    OCoNTemplateEntry item_entry
      = new OCoNTemplateEntry(WarehouseSection
                                   .ITEM_TYPE_ATTRIBUTE_NAME,
                              "TestItem1");
    WarehouseSection section
      = requestManager.getService(WarehouseSection.class,
          template);
    Item item = new TestItem1();
    try {
      section.supply(item);
      System.out.println("Item supplied.");
    }
    catch (NetworkException e){
      e.printStackTrace();
    }
  }

  public void disconnectedFrom(String interface_name){}
}
```

Listing 14 outlines a client implementation that consumes an item. For this purpose it uses an asynchronous service proxy. Directly after the execution of the consume method the client waits for the execution result to arrive with help of the method `waitForInvocationResults(ServiceInterface proxy)`. This method returns if at least one result for the service that is provided as parameter is available. If more than one method has been executed before collecting the result(s) from the `RequestManager` you have to ensure that all results required are collected. If for example five methods have been executed a while loop may be required to wait for all five results to arrive. The class `InvocationResult` provides methods to determine from which method a result is and if the invocation is successful.

Listing 14: A client using an asynchronous service proxy.

```
import ...template.*;
import ...framework.client.*;
import ...example.service.warehouse.*;
import ...com.exception.NetworkException;

public class Consumer implements Client{

  protected RequestManager requestManager;

  public Consumer(){
    this.requestManager = new RequestManager(this);
  }

  public consumeItem(){
    OCoNClientTemplate template
      = new OCoNClientTemplate();
    OCoNTemplateEntry item_entry
      = new OCoNTemplateEntry(WarehouseSection
                                  .ITEM_TYPE_ATTRIBUTE_NAME,
                              "TestItem1");
    WarehouseSection section
      = requestManager.getService(WarehouseSection.class,
          template);
    Item item = new TestItem1();
    try {
      section.consume(item);
      InvocationResult[] results
        = new InvocationResult[0];
      /* wait for results */
      this.requestManager.waitForInvocationResults(section);
      results
        = this.requestManager.getInvocationResults(section);
      if (results[0].getError() == null){
        TestItem1 item = (TestItem1)results[0].getResult();
        System.out.println(item +" consumed.");
      }
      else {
        results[0].getError().printStackTrace();
      }
    }
    catch (NetworkException e){}
  }
```

```
  public void disconnectedFrom(String interface_name){}
}
```

In both implementations the method `disconnectedFrom(String interface_name)` does nothing. Here, handling code can be implemented for the case that the connection to services of the provided `interface_name` gets lost. To start the clients a class with a `main()` method is required. From there the clients have just to be instantiated and the method `supplyItem()` respectively `consumeItem()` has to be called.

If a client shuts down or a reference to a service is not needed anymore, a client is encourage to indicate that to its `RequestManager` by invoking `logOff()` for the former case and `logOff(ServiceInterface proxy)` for the latter.

5.3 Logging

The framework comes with its own logger class that internally uses `log4j`. All logging classes can be found in the logging package `...framework.util.logging`. An alternative logger can be provided by extending the class `Logger` in the logging package. In order to use a new logging implementation the system property `de.uniba.wiai.lspi.ocon.-framework.util.logging.logger.class` has to be set to the fully qualified class name of the custom logger and the custom logger has to be in the classpath of the framework. To get a reference to the logger the `getLogger()` method of the frameworks `Logger` class has to be invoked.

Enabling `log4j` requires that a jar file called `log4j.jar` containing at least `log4j` version 1.2.8 is placed in the `lib` directory of the installation directory. To configure `log4j` a properties file as described in the `log4j` manual can be used. The full path to this file has to be set with the system property `log4j.properties.file`[20].

5.4 Compiling the framework

If the source files of the framework have been installed, it can be compiled with help of the build file (`build.xml`) in the installation directory. To use this build file `ant` is required. To compile the framework the following jar files have to be placed in the `lib` subdirectory of the installation directory:

- `jini-core.jar`: Jar file containing files of Jini v. 2.0_001[21]. Also required to run the framework.
- `jini-ext.jar`: Jar file containing files of Jini v. 2.0_001. Also required to run the framework.
- `sun-util.jar`: Jar file containing files of Jini v. 2.0_001. Also required to run the framework.

[20] Note: On Windows based systems replace the backslashes of the path with slashes.

[21] Can be copied from the `lib` directory of the jini installation.

- `tools.jar`: Jar file containing files of Jini v. 2.0_001. Also required to run the framework.
- `log4j.jar`: Jar file containing `log4j` (at least version 1.2.8). Only required to compile the framework.

Following targets are available from the build file.

- `clean`: Deletes the `classes` and `dist` directories.
- `init`: Creates the `classes` and `dist` directories. Depends on clean.
- `compile`: Compiles the framework. Depends on `init`.
- `rmic`: Creates the RMI stubs and skeletons for the framework. Depends on `compile`.
- `dist`: Creates a jar file containing the framework in directory `dist`. Depends on `rmic`.
- `documentation`: Creates the javadoc documents from the source files in directory `docs/api`.
- `full`: Compiles the framework, creates RMI stubs and skeletons, generates the jar file with the framework and creates the javadoc documents.
- `main`: The standard target. Compiles the framework and creates RMI stubs and skeletons.

References

[BPSM+04] Bray, Tim, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau John Cowan: *Extensible Markup Language (XML) 1.1.* http://www.w3.org/TR/2004/REC-xml11-20040204/, 2004.

[Fou04] Foundation, Apache Software: *Logging Services: log4j version 1.2.8.* http://logging.apache.org/log4j/docs/download.html, 2004.

[FSF91] Free Software Foundation, Inc.: *GNU General Public License.* http://www.gnu.org/copyleft/gpl.html, 1991.

[SM02] Sun Microsystems, Inc.: *Permissions in the Java 2 SDK.* http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html, 2002.

[SM04a] Sun Microsystems, Inc.: *Java 2 Platform, Standard Edition (J2SE).* http://java.sun.com/j2se/index.jsp, 2004.

[SM04b] Sun Microsystems, Inc.: *Jini Network Technology - Sun Community Source Licensing (SCSL).* http://wwws.sun.com/software/communitysource /jini/download.html, 2004.

[WGG97] Wirtz, Guido, Jörg Graf Holger Giese: *Ruling the Behavior of Distributed Software Components. Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada,* 1997.