Design Methods for Distributed Systems

Architecture

VU Entwurfsmethoden für Verteilte Systeme

SS 2008

Group 69

Martina Lindorfer	0626770	066937	e0626770@student.tuwien.ac.at
Florian Lukavsky	0325558	066937	e0325558@student.tuwien.ac.at
Daniel Priewasser	0626777	066937	e0626777@student.tuwien.ac.at
Gerald Scharitzer	0127228	066937	e0127228@student.tuwien.ac.at
Dirk Wallerstorfer	0626775	066937	e0626775@student.tuwien.ac.at

Contents

1 Introduction

1.1 Lab Content & Purpose

In the course of this lab exercise a peer-based middleware based on the design and architecture described in [1] was developed. For demonstration purposes a distributed sales system for an eCommerce company including functionalities for managing customers and products, authorization and ordering was implemented.

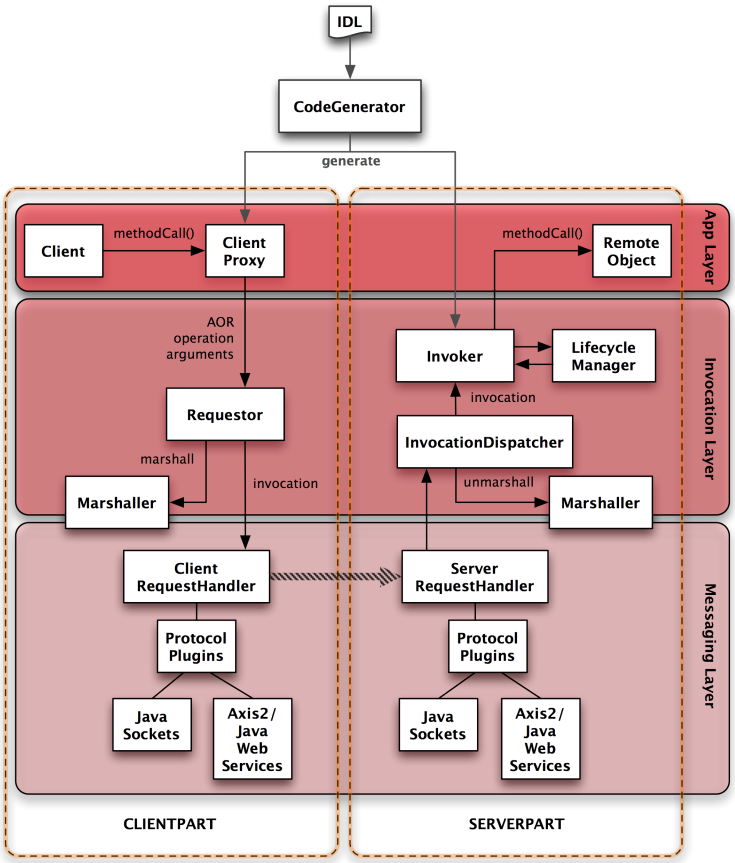

Figure 1: Overall Architecture based on [1]

1.2 Scope of this Document

The purpose of this document is the specification of the architecture of the developed middleware. Additionally a description of the provided distributed sales system as an example use case scenario as well a description of various provided unit tests is given.
The document clearly focuses on architectural aspects and is not intended as a documentation of the source code, functions and parameters. For this purpose a separate Javadoc is provided.

2 Logical View

2.1 Common

2.1.1 Interface Description

The interface description is the source for generating the interfaces for client proxies and remote objects. For the purpose of defining such an interface description the syntax for a simple IDL (Interface Description Language) was created and a corresponding parser was developed. This syntax allows the definition of classes and exceptions in the following form:

```
PACKAGE bla.bla.blub
CLASS Dummy[
Integer tada
void foo(integer a, integer b) THROWS dummyException, IOException
Integer bar() THROWS dummyException
blablablub(Integer c)
]
EXCEPTION DummyException[
Integer count
]
```

The IDL-Parser `evs.idl.SimpleIDLParser` analyzes the file and generates Client and Server Stubs for the specified application. A Code-Generator for Java was implemented in `evs.idl.JavaCodeGenerator`, which generates the following files based on the interface definition:

- An interface for the operations provided by the Remote Object
- The basic structure of the Remote Object implementing this interface
- A Client Proxy implementing this interface for the client-side
- An Invoker implementing this interface for the server-side
- Custom-Type Exceptions

2.1.2 Marshaller

For general marshalling purposes the class `evs.core.BasicMarshaller` is provided. The marshaller translates objects into sequences of bytes and vice versa. Every object that is passed to the marshaller, must at least implement the `Serializable` interface, provided by Java. Otherwise it is not guaranteed that the passed object can be transformed into a byte array. If neither the `Serializable`- nor the `Externilizable`-interface are implemented, a separate marshaller is necessary, that possesses the ability to transform the data of the object into a byte array, despite the lack of the previously mentioned interfaces. The marshaller is implemented as a Singleton, because a peer owns a client and a server interface at the same time and on both sides it is necessary to either marshall or unmarshall a message. Both sides use the same marshaller that offers the required functionality.

2.1.3 Absolute Object Reference

When a peer wants to invoke a remote object, it first needs to know where the remote object is located, hence which server request handler it has to direct the request to. The server request handler is identified by a hostname and a port, where it listens for incoming requests. The next important information the peer needs to know is the ID of the remote object that it wants to invoke, and the ID of the invoker that is behind the prior mentioned server request handler. The pattern *Absolute Object Reference* uniquely identifies the invoker and the remote object. It contains the hostname and the port of the server request handler, the ID of the invoker and the ID of the remote object. If a peer has the AOR of a remote object it has all the information it needs to invoke it.
The AOR is implemented in `evs.core.AOR` and contains the location of the remote object in the form of a `evs.interfaces.ILocation` and the *Object ID* implemented in the `evs.core.ObjectReference`.

2.1.4 ACT

An `evs.core.ACT` is an asynchronous completion token. Clients can start several asynchronous requests at the same time. To give the client the ability to connect an incoming response to a request it sent, an ACT is used. It is passed to the client proxy as a parameter during invocation which passes the ACT further down to the requestor. The requestor serializes the ACT into the request sent. When the requestor receives a response it knows which ACT the answer belongs to and passes it back to the client. The client can now match the incoming response to the request with the corresponding ACT.

2.1.5 Invocation Object

The necessary information for invoking a remote object is bundled in the `evs.core.InvocationObject` which provides an implementation of the `Externilizable`-interface for easy marshalling. This information includes:

- Absolute Object Reference
- the name of the operation
- a list of arguments
- the return parameter
- a remoting exception
- list of invocation contexts

2.1.6 Remoting Errors

It is possible for Remote Objects to declare application-specific exceptions. These exceptions must inherit `evs.exception.RemotingException` and are transported back to the the client, where they are filtered and thrown as application-specific exceptions by the Client Proxy. All distribution-related exceptions are thrown as `evs.exception.NotSupportedException`.
For the internal differentiation of all sorts of distribution-related exceptions by the middleware a range of `MiddlewareExceptin`-subtypes was implemented in the `evs.exception`-package.

2.1.7 Interceptor Architecture

Interceptors can be registered with the `evs.core.InterceptorRegistry` of a peer. Every interceptor must implement the methods `beforeInvocation` and `afterInvocation` of the `evs.interfaces.IInterceptor`. These methods are called before and after the invocation of an object in the requestor on the client-side and in the invoker on the server-side.

2.1.8 Peer

Peers can use one or more client and server request handlers to communicate with other peers. They can create remote objects and make them available to other peers to act as an object server. When acting as a client the peer creates client proxies and invokes their methods to access the remote objects. The peers can be started in separate threads or as main programs. In both cases the actions of the peer are controlled by its input and output streams, which provide a command shell to the user.

2.2 Server-Side

2.2.1 Remote Object

A remote object can be accessed by peers using its Absolute Object Reference. It usually provides some sort of functionality, that the peers want to utilize. For each remote object a corresponding invoker and a client proxy is generated according to the remote object's interface. The client is presented the same interface as the remote object's by its proxy.

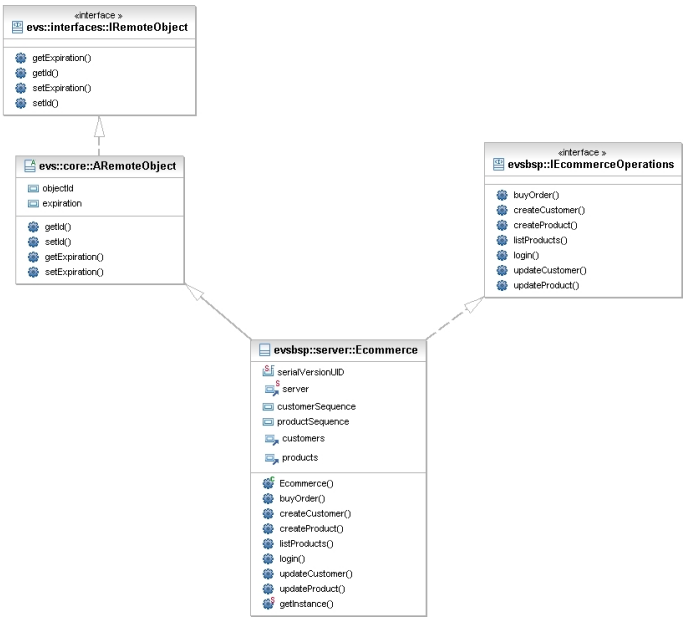

Figure 2: Ecommerce Remote Object

2.2.2 Invoker

For each remote object a corresponding invoker is generated, as mentioned previously. The invoker gets its data for the actual invocation from the `evs.core.InvocationDispatcher`, which unmarshalls the received request and extracts the required information to choose the appropriate invoker. The reason why the invoker is not called directly by the request handler is,

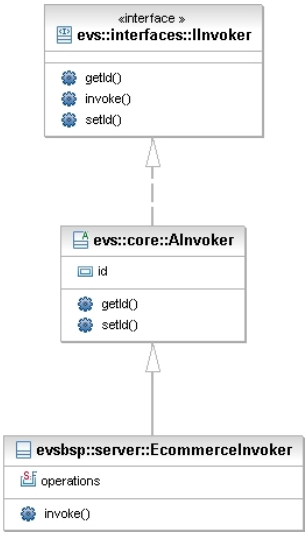

Figure 3: Ecommerce Invoker

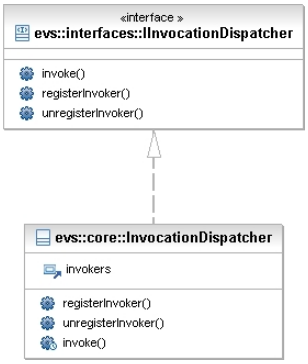

Figure 4: Invocation Dispatcher

2.2.3 Lifecycle Management

For the activation of distributed objects a Lifecycle Manager was implemented in `evs.core.LifecycleManager`. The invoker does not call the object directly but rather receives an object instance from the Lifecycle Manager.
A peer has to register his different types of offered remote objects with a specific strategy with the Lifecycle Manager. Possible strategies include:

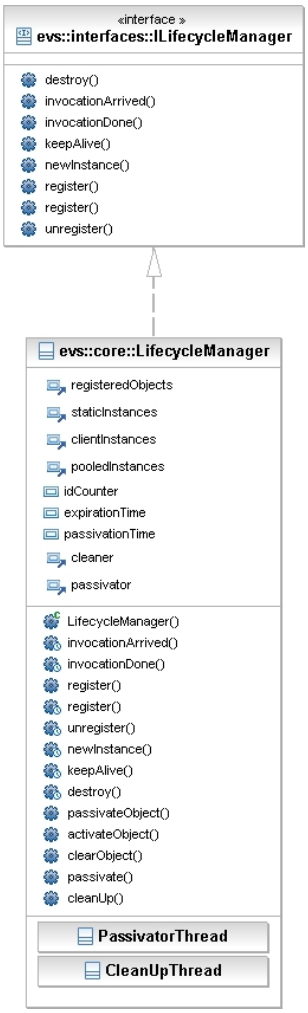

Figure 5: Lifecycle Manager

- `LifecycleStrategy.STATIC`: Static instance, which is activated at registering
- `LifecycleStrategy.STATIC_PASSIVATION`: Static instance, which is passivated to the file system after a pre-configured lease time
- `LifecycleStrategy.PER_REQUEST`: A new servant is instantiated for each for each invocation.
- `LifecycleStrategy.CLIENT`: Client-specific instance (with Leasing)
- `LifecycleStrategy.CLIENT_PASSIVATION`: Client-specific instance, which is passivated to the file system after a pre-configured lease time
- `LifecycleStrategy.LAZY`: Static instance, which is activated at request
- `LifecycleStrategy.POOLING`: Pool of client-independent instances

For the client-specific instances, functions for creating and destroying instances as well as renewing leases has to be provided to the client. All remote objects therefore inherit the abstract `evs.core.ARemoteObject` while all client proxies inherit the abstract `evs.core.AClientProxy` which add lifecycle-specific information to the application-specific objects and proxies.

2.2.4 Server Request Handler

The server request handler receives request messages and dispatches them to the invokers. The server request handler sends response messages to the client request handlers.

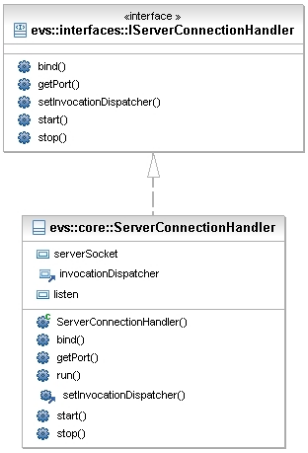

Figure 6: Server Request Handler

2.3 Client-Side

2.3.1 Client Request Handler

The client request handler sends marshalled requests to the corresponding server request handler without interpreting the request. Depending on the invocation style the client waits to receive a response from the server or not. If the a marshalled response is received, then it is returned to the requestor without interpreting it.

2.3.2 Requestor

The requestor receives the following data from the client proxy: an `InvocationObjec`, a boolean indicating, if the called function is `void`, an `ICallback` for the callback functionality of the `InvocationStyle.RESULT_CALLBACK`, an `IACT` (an asynchronous completion token) and the `InvocationStyle` that is used for this invocation. At first the requestor calls every registered interceptor prior to invocation. Next it marshalls the `InvocationObject` to a byte array, in order to prepare it for transmission to the server request handler. Then the AOR (Absolute Object Reference) is extracted from the `InvocationObject`, so that the requstor can tell the request handler explicitly to which location the request has to be sent. In the next step the requestor distinguishes between the different `InvocationStyles`:

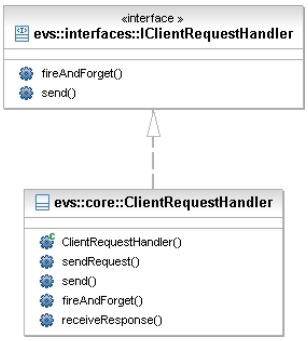

Figure 7: Client Request Handler

- `InvocationStyle.SYNC` In the case of `SYNC` the request handler is called, and the requestor waits for its answer. The received result is unmarshalled and then returned to the client proxy.
- `InvocationStyle.POLL_OBJECT` When using a `POLL_OBJECT` the requestor creates a new `PollObjectRequestor`, which takes responsibility for the further handling of the request. The newly created object starts in a new thread, marshalls the `InvocationObject`, creates a new `PollObject`, sends the request to the request handler and waits for a response. The created `PollObject` is returned to the client proxy, which passes it further on to the client. Now the client can poll this object, if the result of the invocation has arrived yet. When the result arrives, the requestor gets the marshalled response from the request handler, unmarshalls it, saves the response in the `PollObject` and sets a boolean to true, to reveal that the response has arrived.
- `InvocationStyle.FIRE_FORGET` The style `FIRE_FORGET` is quite simple. The request is passed to the request handler, which sends the request to the desired location. The request handler does not wait for an answer, neither do the requestor, the client proxy or the client.
- `InvocationStyle.RESULT_CALLBACK` The `RESULT_CALLBACK` style implies, that after the request is sent, no one waits for a response. When the response arrives, the request handler calls a callback function to notify the requestor about it. In order to achieve this functionality, the requestor has a `Map<IACT, ICallback>` to keep track of the request and response messages. Each call is saved as a new entry in the map. The requestor creates a new `ResultCallbackHandler` object, which runs in its own thread. The request is ordinarily sent via the request handler. When the `ResultCallbackHandler` gets the response from the request handler, it calls a function in the requestor and passes the result as a byte array and the `ACT` (asynchronous completion token). The requestor unmarshalls the response, fetches the corresponding `ICallback` to the passed `ACT` and calls the `resultReturned` function of the `ICallback` with the response as parameter.

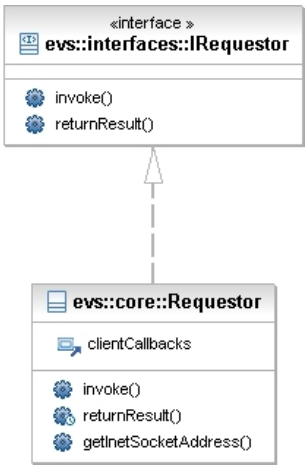

Figure 8: Requestor

2.3.3 Client Proxy

The client proxy implements the same interface as the remote object, so the client calls the client proxy as it would call the remote object directly. The proxy is generated automatically for each remote object. The client proxy handles the different `InvocationStyles`. Independent of the chosen `InvocationStyle` the client proxy forwards the invocation to the requestor. It creates a new `InvocationObject` from the invocation of the client which is passed to the requestor. Furthermore a boolean, indicating wether the method called, has a return value or not, an `ICallback`, where the callback for the `InvocationStyle.RESULT_CALLBACK` is set or not, an ACT (Asynchronous Completion Token) and the `InvocationStyle`, that was set prior to the invocation, are passed. The requestor now has all the information it needs for further processing.

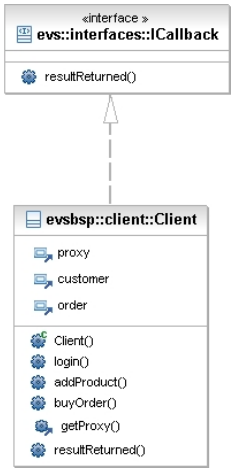

Figure 9: Ecommerce Client

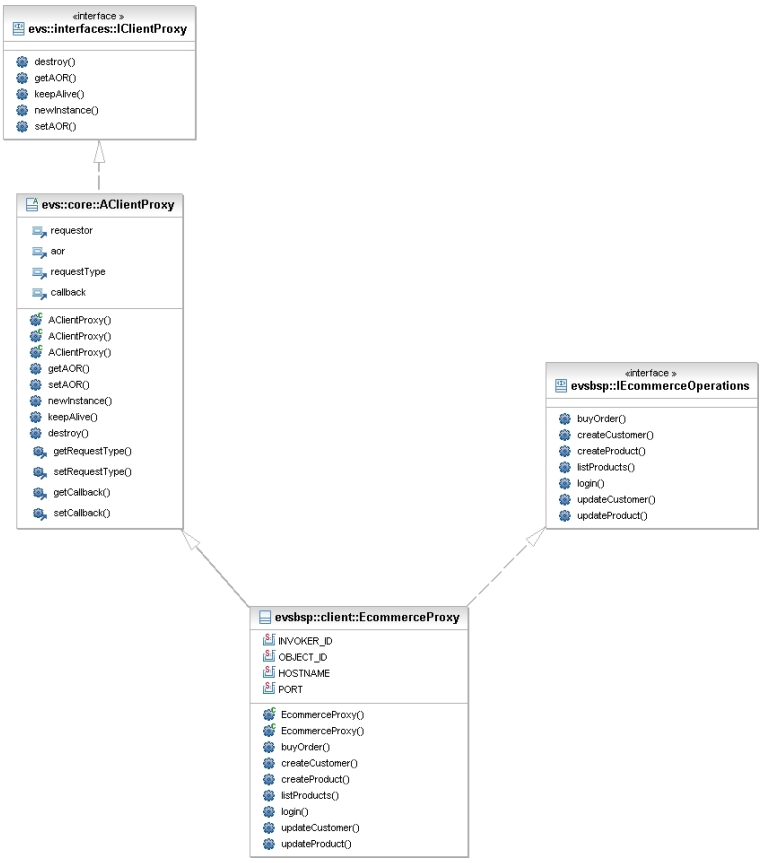

Figure 10: Ecommerce Proxy

2.4 Protocols

2.4.1 Communication Protocols

The communication protocol is determined by the communication end points, which process only single protocols.

Basic Communication Protocol The basic communication protocol uses message headers but transmits byte streams without interpreting them. The sender and receiver are determined by the communication end points.

Axis2 Java Web Service Protocol The client request handler invokes operations of an invoker service which is provided by.

2.4.2 Marshalling Protocols

Java Serialization Protocol The invocation is serialized by the default Java object serialization.

3 Development View

3.1 Tools

Development Environment	Eclipse 3.3.2	`http://www.eclipse.org/`
Programming Language	Java 5	`http://java.sun.com/`
Web Services	Axis2 1.4	`http://ws.apache.org/axis2/`
Unit Tests	JUnit 4	`http://junit.org/`
Build Tool	Ant 1.7	`http://ant.apache.org/`
Version Control	Subclipse 1.2.4	`http://subclipse.tigris.org/`

3.2 Naming Rules

- The top level package is "evs" (for the middleware implementation) or "evsbsp" (for the use case implementation) and there is only one level of sub packages.
- Interface names start with "I".
- Abstract class names start with "A".

3.3 Packages

`evs.axis`	Axis2 request handling
`evs.core`	communcation framework
`evs.exception`	remoting exceptions
`evs.idl`	interface definition language
`evs.interfaces`	interfaces for communication framework
`evs.main`	main classes
`evsbsp.client`	client-part of the use case scenario
`evsbsp.entities`	business entities of the use case scenario
`evsbsp.junit`	JUnit tests
`evsbsp.server`	server-part of the use case scenario
`dummy.*`	simple use case for unit testing the middleware

4 Process View

The communication is based on Java sockets where each communication end point is identified by its IP address and TCP port number.

4.1 Request Handlers

The request handlers support two different communication protocols.

4.1.1 Socket

For each request the client request handler creates a new connection to the server request handler. Every message is prefixed by a message header, which contains the length of the serialized request and the invocation style. After the message header the message body is sent, which consists of the serialized request. If the invocation style is fire and forget, then the client closes the socket without waiting for a response from the server. Otherwise the client request handler waits to receive the message header of the response message. Finally the client request handler receives the serialized response and returns it to the requestor.

For each request the server connection handler accepts a new connection and starts a service request handler in a separate thread. The server request handler waits to receive the message header and the serialized request. The request is then passed to the invocation dispatcher, which returns the serialized response. If the invocation style is fire and forget, then the server closes the socket without sending a response. Otherwise the server sends a message header, which contains the length of the serialized response. Finally the server request handler sends the serialized response.

4.1.2 Axis

For each request the client request handler creates a web service invocation of the invoker service. If the invocation style is fire and forget, then the serialized request is sent to the operation "fireAndForget", which does not return a result. Otherwise the serialized request is sent to the operation "invoke" and the response is returned to the requestor.

The server request handler is an instance of an Axis server and provides an invoker service. This invoker service provides the operations "invoke" and "fireAndForget", which receive the serialized requests. The invoke operation passes the request to the invocation dispatcher and returns the serialized response. The fireAndForget operation passes the request to the invocation dispatcher without returning a response.

4.2 Message Patterns

The request handlers support two different message patterns.

4.2.1 Request Response

The client request handler sends the serialized request to the server request handler and waits for the response. The server request handler receives the serialized request

and passes it to the invocation dispatcher. The invocation dispatcher returns the serialized response and the server handler sends it to the client handler.

4.2.2 One Way

The client request handler sends the serialized request to the server request handler without waiting for a response. The server request handler receives the serialized request and passes it to the invocation dispatcher. The result of the invocation is not sent to the client.

4.3 Invocation Styles

The requestors support one synchronous and three asynchronous invocation styles.

4.3.1 Synchronous

The requestor synchronously invokes the send method of the client request handler and waits for the result.

4.3.2 Poll Object

The requestor creates a new poll object and passes it to a poll object requestor. The poll object requestor is started in a separate thread and the requestor returns the poll object to the client proxy. In the poll object requestor thread the invocation is marshalled and the send method of the client request handler is invoked synchronously. The response from the client request handler is marshalled and stored in the poll object.

4.3.3 Result Callback

The requestor creates a new result callback client request handler and passes an ACT to it. The result callback handler is started in a separate thread and sends the serialized request to the server request handler. After receiving the serialized response the result callback handler invokes the resultReturned method of the requestor with the ACT and the response. This resultReturned method looks up the callback for the ACT and invokes the callback with the unmarshalled response.

4.3.4 Fire and Forget

The requestor invokes the fireAndForget method of the client request handler and does not receive a result.

4.4 Peers

The peers can start one or more server request handlers in separate threads to act as servers without interfering with their client functionality.

5 Scenarios

5.1 Use Cases

5.1.1 Manage Customers

The server stores customer objects in memory. To manage these customers, the server provides a create and an update function. The customer's ID is generated by the server and is the only parameter of a customer, that may not be changed, in order to allow the server to uniquely identify the desired customer. Customer's name, username and password, in contrast, may be changed. Additionally the customer object provides functionality to keep record of all orders, placed by this specific customer.

5.1.2 Manage Products

The management procedures of products are similar to those of customers. Products are stored in memory and can be manipulated with a create and update function. As with the customer, the server allocates a unique ID to each newly created product. Furthermore a product is described by its name and its price.

5.1.3 Customer Login

In order to place an order, the customer has to identify itself with its username and password. If the login is successful, a `Customer` object is returned, holding the customer's data, and all its placed orders.

5.1.4 Find Product

The server provides functionality to obtain all available products. The customer can pick some of these and add them to an order, which he can decide to buy.

5.1.5 Buy Order

After the customer has successfully logged in and decided which products to add to the order, he can call the servers buy order method, sending the generated order object and attached to it, the customer's object. The server adds the order to the customer whos ID corresponds to the ID received by the server, making it accessible in later sessions.

6 Configuration

The file `conf/evs.properties` contains configurable parameters of the middleware implementation. These include the location of the peer as well as lease time, pool size, passivation timeout and passivation directory for the lifecycle management.

7 Conclusio

The developed middleware is based on the design and architecture as described in [1]. Additionally the middleware uses most of the basic Remoting Patterns shown during the evs lecture in the summer term of 2008. To be more accurate, the so called `minimal solution` presenting the communication framework is implemented. As part of this framework we provide the following functionality:

- Each peer uses a RequestHandler, providing the functionality of the Client- and Server-RequestHandler.
- Remote invocations are abstracted using a Requestor and Invoker.
- A Client Proxy shares a common interface with the Remote Object and maps its methods to calls to the Requestor's invocation interface.
- The remote invocation data consists of 1) methods and parameter 2) an Object ID 3) an Absolut Object Reference.
- Flexible protocol implementation is provided using Protocol Plugins.

The enhanced capabilities of the middleware, as described in extension part 1, are provided with some shortcomings: We do not provide the federation approach using Configuration Groups. Except this the middleware implements the following client-side asynchronous invocations: Fire and Forget, Poll Object, Result Callback. The implementation of an interceptor architecture based on Extension Patterns and an architecture for activation of distributed objects, using Lifecycle Managemnet Patterns is supported by the middleware.
The functionality of dynamic deployment of new services is provided by the implemented Peer. The Peer configuration and invocation is implemented using Java instead of Frag. To demonstrate the correct behavior of the presented middleware we provide the following provisions:

- The middleware is used to implement a distributed sales system for an eCommerce company.
- For this eCommerce company we provide corresponding JUnit tests.
- JUnit test for parts of the middleware are also provided.

A Abbreviations and Acronyms

ACT	Asynchronous Completion Token
AOR	Absolute Object Reference
IDL	Interface Description Language
IP	Internet Protocol
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol

B References

[1] Uwe Zdun. Pattern-Based Design of a Service-Oriented Middleware for Remote Object Federations. 2008.
`http://www.infosys.tuwien.ac.at/staff/zdun/publications/leelaSOAMiddleware.pdf`.