VamPeer

JXTA implementation for Eiffel

Beat Strasser `<beat@stradax.net>`

Master thesis

September 2006 – March 2007
ETH Zürich, March 3, 2007

Supervisor: Till G. Bay `<till.bay@inf.ethz.ch>`

Professor: Bertrand Meyer `<bertrand.meyer@inf.ethz.ch>`

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chair of
Software Engineering

Abstract

The goal of this master thesis is to equip Eiffel with a peer-to-peer framework. The well-known JXTA protocol was chosen for that purpose.
VamPeer aims to be an Eiffel binding for JXTA, currently offering the essential services used in a typical edge peer. It thus allows to discover remote peers and to communicate with other peers through a TCP transport. Implementing the discovery service, it is possible to query for local and remote entities. The library is fully compatible with the latest JXTA JSE reference implementation.

Contents

1 Introduction **9**

1.1 Mission . . . 9

1.2 Related work . . . 10

1.3 Chapter overview . . . 10

2 JXTA **11**

2.1 Peer groups . . . 11

2.2 IDs . . . 12

2.3 Advertisements . . . 13

2.4 Services . . . 14

2.4.1 Endpoint service . . . 15

2.4.2 Transport modules . . . 16

2.4.3 Rendezvous service . . . 17

2.4.4 Resolver service . . . 19

2.4.5 Discovery service . . . 20

2.5 JXTA's P2P infrastructure and peer roles . . . 21

3 Design **22**

3.1 Requirements . . . 22

3.2 Module structure . . . 23

3.2.1 Peer group modules . . . 24

3.3 Defining a peer group . . . 25

3.4 Services . . . 26

3.4.1 Module choice . . . 26

3.4.2 Service layers . . . 27

3.5 Address rewriting . . . 29

3.6 Rendezvous propagation . . . 30

4 Implementation **31**

4.1 Development environment . . . 31

4.2 Used libraries . . . 31

4.3 Socket extensions . . . 32

4.4 XML documents . . . 33

4.5 Using UUID for JXTA IDs . . . 34

4.6 Threads . . . 35

4.6.1 TCP Transport . . . 36

4.6.2 Rendezvous connection manager . . . 37
4.6.3 Discovery SRDI . . . 37
4.7 Advertisement storage . . . 37
4.7.1 Persistent storage . . . 37
4.7.2 LRU cache . . . 39
4.8 Shared creators . . . 39

5 Usage **41**
5.1 Platform starting . . . 41
5.1.1 Private peer groups . . . 42
5.2 Services . . . 45
5.2.1 Endpoint service . . . 45
5.2.2 TCP Transport module . . . 48
5.2.3 Rendezvous service . . . 49
5.2.4 Resolver service . . . 51
5.2.5 Discovery service . . . 52
5.3 Writing a P2P application . . . 54
5.3.1 Writing a user service . . . 55
5.4 Examples . . . 56
5.4.1 Endpoint message sender/handler . . . 56
5.4.2 Rendezvous propagation . . . 57
5.4.3 Discovery . . . 57
5.4.4 JXTA JSE rendezvous server . . . 57

6 Origo with VamPeer **58**
6.1 Origo overview . . . 58
6.2 Design . . . 59
6.2.1 Task . . . 59
6.2.2 P2P infrastructure . . . 60
6.2.3 Startup procedure . . . 61
6.2.4 Role configurations . . . 61
6.3 Summary . . . 61

7 Results **62**
7.1 Benchmark . . . 62
7.2 Unit tests . . . 62
7.3 Summary . . . 63

8 Conclusions **65**
8.1 Future work . . . 65
8.2 Acknowledgements . . . 66

List of Figures

3.1 Module life cycle 23
3.2 Module class hierarchy 24
3.3 Information flow for an outgoing discovery query 27
3.4 Information flow for an incoming discovery response 29
3.5 A mangled service handler name 30

4.1 XML document class hierarchy 33
4.2 ID class hierarchy 34

6.1 Origo example P2P class structure 60

List of Listings

2.1 A sample peer advertisement for a peer in the public NPG 14
2.2 A sample resolver query XML document 19
3.1 Clusters overview 26
4.1 Persistent storage directory layout 38
5.1 Configuring the platform instance 41
5.2 Loading/starting the platform with the public NPG 42
5.3 Creation of IDs for a new peer group 43
5.4 Creating a platform configuration for a private peer group 43
5.5 Creating a peer group module implementation advertisement 44
5.6 Loading a private NPG 44
5.7 Example endpoint message handler 46
5.8 Creating and sending an endpoint message 47
5.9 Example endpoint message filter 48
5.10 Full TCP transport configuration 48
5.11 Example rendezvous event handler 50
5.12 Public NPG rendezvous seeds 50
5.13 Example rendezvous configuration 51
5.14 Sending a resolver query 51
5.15 Sending a remote discovery query 53
5.16 Publishing an advertisement remotely 54
5.17 Redefining peer group modules 55
7.1 Benchmark output 62
7.2 Unit test results 63

List of Tables

2.1 UUID ID types in JXTA IDs 13

3.1 Required services 23

1 Introduction

Peer-to-peer systems have become very popular in the last few years. They allow users to share resources (such as calculation power or information) in a distributed and decentralized way. Peer-to-peer (henceforth called P2P) technology firmly differs from the client-server model which relies on one central server fulfilling all tasks for the clients whereas in a P2P system every participant is considered equivalent. [MKL02] gives a good overview of P2P based on the most important systems and summarizes the key concepts.

In P2P applications, we often want to perform the same tasks like discovering other peers, sending and propagating messages and sharing information. While many P2P applications implement their own solutions, there also exist frameworks which provide a simple API for most of these common tasks.

1.1 Mission

Unfortunately for the Eiffel programming language, there was no such framework. This master thesis fills that gap and implements an existing framework: JXTA. We call our library *VamPeer*.

The main purpose of our implementation is to have a P2P library available in Eiffel so that it can be used in Origo, a new software development platform by Till Bay. The Origo platform is a distributed and very modular as well as an extendable system integrating the usual facilities like version control, bug tracking and project web hosting together with a single sign-on solution[1]. It will use our P2P library so that the various modules, each residing on another peer, may seamlessly communicate with each other.

Chapter 6 gives a more detailed introduction to Origo. For now, we would like to point out that we only supply the P2P library but do not integrate it into Origo. Our library is generic in so far as it can be used for many purposes and is not coupled to the Origo platform with respect to the code. But still, our long term goal is to satisfy Origo's networking needs with a library.

JXTA is an open source P2P framework created by Sun Microsystems [Pro]. It is one of the most mature platforms in its field. JXTA (pronounced *juxta*) is composed of several modules each implementing a JXTA protocol (for example the discovery protocol). The protocols are heavily based on XML.

[1] *Single sign-on* is an authentication technique allowing a user to authenticate once in order to gain access to several resources.

1.2 Related work

In Eiffel, there is currently no P2P library available. As far as we know, we are the first building such a library. Fortunately, there is a number of general networking libraries so that we do not have to start from scratch.

The decision to use JXTA was quite simple since it is the only platform independent framework. This gives us the advantage to port it to Eiffel and to be compatible with other implementations. While the reference implementation is written in Java JSE, there also exist other bindings written in C, JXME and others. Most of the other smaller bindings are not yet ready to use.

Probably one of the most alike project is called *Jini*. It is unsuited for our purpose since it only runs on Java. Furthermore, it uses a central server to locate network services in contrary to JXTA which follows a completely decentralized P2P model.

There is also another framework named *OogP2P* but since it is a simple study project and unmaintained since several years, we did not have a closer look at it.

Besides frameworks, there are plenty of P2P networks defining a full protocol for sharing content (for example *GnuNet*). Research projects such as *Chord* usually provide special algorithms for a distributed hash table. These projects do not meet our demands since they focus too much on sharing information and lookup algorithms instead of more general P2P facilities.

We therefore build an Eiffel binding for JXTA because it gets used more and more in today's applications (for example *Collanos Workplace*). A short overview of the mentioned P2P frameworks and protocols is available at the *VamPeer*'s website[2].

1.3 Chapter overview

We now proceed to present our new library dealing with the following topics:

Chapter 2 introduces the JXTA protocols we are dealing with and explains JXTA's key concepts. In chapter 3, we analyze the requirements for Origo and then define how we designed our library to comply with the JXTA standard. Chapter 4 deals with the implementation and demonstrates how we mastered the challenges. The subsequent chapter 5 lists the *VamPeer*'s possibilities and explains the library API based on some examples. A larger example is shown in chapter 6 where we demonstrate how messaging could be done in Origo. Chapter 7 covers the results of our work and its performance. We finally conclude with chapter 8 where we will have an outlook to future work.

[2]The project is hosted on Origo at: `http://origo.ethz.ch/index.php/VamPeer`

2 JXTA

Before we describe the *VamPeer* design, we give an introduction to the JXTA world. We show how the JXTA protocols are designed and present a specifications overview.

The full specification [Pro07], which is available online, mainly covers the basic ideas around JXTA and specifies nearly only the messages which go over the wire. For more semantical details, we recommend further literature or to look at the reference implementation source code.

The paper [TAD03] focuses on the newer release 2 of the JXTA's protocols and gives a good overview. The free book by Brendon Wilson [Wil02] explains JXTA with many Java examples whereas [BGKS02][1] goes more into the details of the Java reference implementation. Unfortunately, the books are slightly out of date.

We will first look at peer groups, define JXTA IDs and advertisements and then introduce the different services and protocols available. Afterwards, we present the overall P2P network infrastructure. We do not remain only at the JXTA specification level but go further and show a few design ideas used in JXTA JSE, the Java reference implementation[2].

2.1 Peer groups

A peer group is a compound of peers agreeing to run the same set of services[3]. When a peer joins a peer group, all services needed should be loaded according to the peer group's specification. Thus, a peer is always a member of at least one group as there would not be any running services at all otherwise.

A peer may belong to more than one peer group though. The super peer group which is loaded first is *usually* the world peer group (WPG). All other groups are direct or indirect children of the WPG. This is because only one peer group can actually handle the network traffic. The specification does not explicitly mention a parent-child relationship among peer groups but it is handled that way in JXTA JSE and also in *VamPeer*.

[1] `www.samspublishing.com` provides a free sample chapter: "Java Implementation of JXTA Protocols"

[2] JXTA JSE is currently available in version 2.4.1, see [Pro]. The next release 2.5 follows in March 2007.

[3] A service is a set of features following a specification either made by the JXTA project or the user, see section 2.4.

World Peer Group The WPG is defined as the peer group in which all JXTA peers reside, even if they are not communicating with each other. The WPG is somewhat a special peer group which is automatically loaded and may not support all services. In the C implementation, there is actually no explicit WPG whereas *VamPeer* uses one for configuration purposes only.

The WPG's ID[4] reads `urn:jxta:jxta-WorldGroup`.

Net Peer Group The WPG's only direct child is the net peer group (NPG) which is now a true normally running peer group configured with all JXTA services.

When talking about the NPG, we usually mean the *public* net peer group with the ID `urn:jxta:jxta-NetGroup`. It should only be used for development and testing purposes (as long as no other group has been created). Sun provides a public infrastructure for this peer group—unfortunately, the servers have been unavailable or under heavy load for the last couple months[5].

When using JXTA as a framework for a custom P2P application, one should create a new *private* NPG so that the application will not get in contact with peers from other applications[6]. It has its own network.

Other peer groups are children of the NPG. Although services can be shared among peer groups, one should have good reasons to split the application into several groups as peer communication is only possible within the same group.

2.2 IDs

We already mentioned IDs for peer groups. Also other entities in JXTA have an ID: peers, modules, advertisements and other resources. A JXTA ID must be a complete identifier referring to a unique resource.

JXTA IDs are in URN format (see [Moa97]) with the namespace `jxta`. Additionally, the URN namespace specific string is prefixed with a format ID announcing how the ID is formatted. The general form looks like this: `urn:jxta:format-specificid`

Although the format is written explicitly, one should never make any assumptions about the ID format. Of course, it is allowed to optionally gather some information from an ID when the format is recognized.

UUID format Most IDs are in the JXTA `uuid` format. They are in hexadecimal form representing 1 up to maximal 64 bytes. The last byte (the last two hex digits)

[4]See section 2.2 for an introduction to JXTA IDs.

[5]This downtime being cumbersome, it is no problem as one can easily setup an own rendezvous/relay server, see subsection 2.4.3.

[6]This is not a guarantee however; every JXTA peer may connect as long as it knows the address and has a purpose.

always specifies the type. Each type stands for its own bytes schema. Usually, the number contains one or two UUIDs[7], each 16 bytes long.

Table 2.1 shows the defined types for the `uuid` format and all the contained information.

Sample IDs look like this:

- Peer group ID:
  `urn:jxta:uuid-822A7C9E6B804759870B81B10070E9C9\`
  `59616261646162614A7874615032503302`
- Module class ID:
  `urn:jxta:uuid-261F502615134AA99FDC99E3751E6B8505`

ID byte	*Type name*	*Information contained*
01	Codat ID	Group UUID, Codat UUID, Codat Hash
02	Peer group ID	Group UUID, Parent group UUID
03	Peer ID	Group UUID, Peer UUID
04	Pipe ID	Group UUID, Pipe UUID
05	Module class ID	Module UUID
06	Module specification ID	Module class UUID, Specification UUID

Table 2.1: UUID ID types in JXTA IDs

2.3 Advertisements

Another important and widely used term in JXTA is *advertisement*. It is an XML document advertising any kind of resource. It does not actually contain the real data, but only metadata. The JXTA protocols are then used to transport and share advertisements with other peers.

For peers, peer groups and all the other JXTA entities, there are advertisement schemes defined. It is the JXTA way of creating new types of advertisements for each used entity. In a file sharing application for example, one would possibly create a file advertisement containing a codat ID[8], the owner's name, the creation date and the peer's ID that hosts the file.

An advertisement has an expiration time, so no old advertisements may be passed around. As advertisements cannot be withdrawn or deleted on remote peers, one should not set the expiration time too high; the default expiration is two hours, but

[7] A "Universally Unique Identifier" (UUID) is a random number (containing also time information) meant to be universally unique (the probability to create two same UUIDs in the same context is very small). They are fully defined in [ISO04]. You may more like the corresponding RFC [LMS05] which describes also the UUID format but focuses more on UUID as a URN namespace.

[8] A codat is just a container for any kind of data, for example file content.

implementations may vary. When resources are valid for a longer time than the expiration time, the advertisement has to be recreated and published again.

We would like to show two advertisement types so that you may better understand its purpose:

Peer Advertisement One of the most important advertisements is the one describing a peer. This is used for example to discover peers. It contains the peer ID, the peer group ID and an optional peer name as well as a description. Each service may additionally add a service parameter with some special configurations.

Listing 2.1 shows a sample peer advertisement. The service parameter hosts the configuration for the endpoint service: a route advertisement that advertises the physical endpoint address (IP host and TCP port).

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PA xmlns:jxta="http://jxta.org">
  <PID>urn:jxta:uuid-59616261646162614E50472050325033\
    1A227980E5924E80A3FD8ECD73D4C31803</PID>
  <GID>urn:jxta:jxta-NetGroup</GID>
  <Name>My sample peer node</Name>
  <Desc>Development test peer</Desc>
  <Svc>
    <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000805</MCID>
    <Parm><jxta:RA><Dst>
      <jxta:APA><EA>tcp://129.132.105.170:32725</EA></jxta:APA>
    </Dst></jxta:RA></Parm>
  </Svc>
</jxta:PA>
```

Listing 2.1: A sample peer advertisement for a peer in the public NPG

Peer Group Advertisement The peer group advertisement announces the existence of a group. It contains the group ID, the module specification ID[9] and optionally again a name, description and service parameters.

2.4 Services

Until now, we have just introduced *peer groups* and the general terms *ID* and *advertisement*. In this section, we will look at *services* and what JXTA protocols they are providing.

JXTA is very modular. Each feature or protocol is available as a service module. Thus, a peer group can disable unneeded modules.

[9]See section 2.4 for an introduction to modules and their specifications.

Module definition JXTA modules can be loaded dynamically. This means that a JXTA peer could theoretically load module code from another peer. For specifying and identifying modules, there exist several module advertisements:

A class of modules providing the same local behavior and API is identified by a unique *module class ID* (MCID) and announced with a *module class advertisement*.

Specifications for a module class are identified with a *module specification ID* (MSID). Its *module specification advertisement* includes a version number and a URI where a human-readable description can be found. The specification focuses on the remote behavior and the protocol.

All modules implementing a module specification can be advertised with a *module implementation advertisement*. This specifies the targeted environment and it may provide the entire code or only a package name and a description where the code can be fetched.

The three module definition layers allow to have various specification versions for a single module class and also any number of module implementations for each environment.

We will now present the main services we have focused on in *VamPeer*:

2.4.1 Endpoint service

The endpoint service is actually the core service where messages are redirected to when coming from the network (through the transport modules) and where messages can be sent to other peers. For the received messages, its job is therefore to route the messages to the services that are interested.

Endpoint messages The endpoint service is dealing with *endpoint messages*. Each transport module must be able to send them over or read them from the wire.

A message is basically just an ordered list of key/value pairs (elements). The key is restricted to a namespace whereas the empty namespace and the `jxta` namespace are predefined for user respectively JXTA internal purposes. You may have any number of namespaces and keys[10].

An element may additionally define a MIME type[11] and a signature element which is rarely used though.

The endpoint service adds some elements to outgoing messages, for loopback detection and for addressing.

Endpoint addresses The message's source and particularly the destination address must somehow be defined: an *endpoint address* can be used for various forms of

[10]There are however some restrictions by the transport modules but beyond of what you will ever need...

[11]See [FB96] for a general introduction to MIME types and [MLK01] for the XML type, which is used very often in JXTA message elements.

addresses. The string format looks like this:

```
protocol://address/service/param
```

The *protocol* part specifies the transport module to use: `tcp`, `jxta`, `http`, ... The *address* is protocol specific: For TCP and HTTP for example, it is of the form: `ip:port`, for JXTA it is simply a peer ID in URN format.

For destination addresses, *service* defines the final service; *param* is an optional parameter for the given service.

Thus to send messages to a peer, one may either create a destination endpoint address directly with the destination's IP and port number or alternatively just set the peer ID, which is the preferred way (because dealing with IP addresses is discouraged in higher level services). It is the *endpoint router*'s task to resolve the peer ID to the real endpoint address as described in the next section.

2.4.2 Transport modules

The *transport modules* are responsible for sending endpoint messages to another peer and for reading incoming messages from the network. Therefore, they register themselves as available transport module in the *endpoint service*, each for its own *protocol*. They provide support for sending a single message to a peer, to ping a peer (looking if the remote peer is online) or to propagate a message[12].

Each transport specifies its own wire representation.

There is no required transport module and protocol but the low level transport TCP is usually enabled together with the HTTP transport. It would also be possible to send messages via SMTP or anything else. Whereas the TCP transport is simple and fast, HTTP has the great advantage to break firewalls since many firewalls allow HTTP traffic.

The transport modules do not guarantee message delivery even when TCP is used. This is very important. The original message sender cannot be sure that his message has arrived at the destination.

The message transport is usually not secured except for the `jxtatls` transport which uses TLS to encrypt data. It is based on top of the *endpoint router* to provide a secure path from the source to the destination peer.

Endpoint Router A special transport is the *endpoint router*. It is not used to transport messages over the wire but to route messages with a peer ID as destination address to the correct gateway. For this, it rewrites the destination address and passes the messages again to the endpoint service, which is then able to send the message using a real transport.

A message cannot be sent directly to the peer because there is no direct connection to it. So called *router peers* may then forward messages to other networks. This

[12]Propagating is only available with UDP multicast.

module's task is to query for routes and to send the message to the first route gateway. It will try to connect using the fastest transport module around (whenever a connection to that peer is already opened, the related transport is considered fast).

Whenever the module, respectively the peer, is configured as *router*, it will accept and forward messages from other peers. It would also maintain a route cache to be able to do its job faster (without always having to first seek for routes).

2.4.3 Rendezvous service

The *rendezvous service* is used for propagating messages through the peer group and/or the local network. This is a very fundamental service and many other services rely on it (e.g. for sending queries to all peers).

The module has basically two modes: `server` and `client`. An edge peer would only implement and run as rendezvous client.

Rendezvous lease protocol A rendezvous client has to subscribe to a rendezvous server to be able to send messages for propagation and also to receive propagated messages. Therefore, a lease protocol is defined which manages this.

When the client peer joins a peer group, it tries to connect to a rendezvous server. There are several ways to find one:

- The peer's platform configuration contains some endpoint addresses (addressing the peer directly with TCP or HTTP). Such addresses may also be hard coded in the user's peer application.
- The peer's platform configuration specifies a seeding URL where a rendezvous server list is published. This is actually the most common and simple way. It is also very useful for maintenance because only one list has to be updated to point all the new peers to other servers.
- As soon as a peer got contact with other peers, it may send them discovery queries for rendezvous peers. It may also cache rendezvous advertisements[13], so it does not have to find new servers when starting up every time.
- Maybe the chance is high that other peers reside in the same local network. Therefore, a peer may send a discovery query via multicast to the local net.

Once a rendezvous[14] gets a lease request, it may send back a lease granted message—a lease which is always restricted for a certain amount of time, usually 30 minutes. During this time, messages are also propagated to the subscribed peer which is allowed to send a propagation message to the rendezvous.

[13]A *rendezvous advertisement* promotes a peer's rendezvous server capability.

[14]Speaking about a *rendezvous*, we always mean a rendezvous server or the service depending on the context.

As leases are not eternally valid, a client has to send lease renewal requests until it gets another lease. Renewal and initial lease requests actually do not differ. JXTA JSE asks for lease renewal when the lease's first half has passed.

When a peer leaves a peer group, it should send a lease cancel message so the rendezvous does not try to propagate messages to that peer anymore.

A rendezvous client should only be registered with at most one rendezvous and should always send a lease cancel message to peers which send propagation messages without being the used rendezvous.

For further details on lease messages, please consult the specification [Pro07].

Message propagation protocol The *message propagation protocol* is used to propagate messages. It adds a message element[15] which has an XML document containing a unique message ID, a TTL, a path and the final destination service's name and parameter.

The protocol makes sure that duplicated messages get discarded as well as messages that are too many hops away from the source peer. This can be done by means of the message ID and the TTL respectively. Every hop decrements the TTL value, so the message can be filtered out when the value reaches zero.

The protocol is also responsible for loop detection. Every hop adds its peer ID to the message's path and detects when a message already passed by earlier.

Based on the given service name, the rendezvous service is able to pass the message to the correct service, for example by using the endpoint service.

Unlike older JXTA versions, a rendezvous service should not repropagate every incoming message and thus flood the network. Each service decides individually whether to repropagate a message or not—of course only as long as the peer is a rendezvous server.

Hence, a rendezvous server should have some knowledge about the network and which peer may have which information. It is able to direct messages only to those peers that may have use for the propagated message. Of course, this heavily depends on the actual service and the message's type.

Peerview protocol Rendezvous servers need as mentioned earlier a good knowledge of the peer network infrastructure. They also need to stay in contact with other rendezvous servers to share propagated messages because there may be any number of rendezvous (not just one) and propagated messages are expected to reach eventually every node in the peer group (not just the subscribed peers of one's own rendezvous).

To manage and share this knowledge, the *peer view protocol* is used. Nowadays, only rendezvous servers should actually run it.

[15] The element name is the peer group ID prefixed with `RendezVousPropagate`.

2.4.4 Resolver service

The *resolver service*[16] is our first user of the rendezvous service as it has to propagate queries. Its task is to provide a query-response system.

It is able to recognize received responses to a sent query by adding meta information to queries like a handler name and a query ID. It attaches also the peer's route information, so remote peers have the possibility to respond even if they do not know the querying peer. The actual query can be any string.

The resolver service creates an endpoint message and combines the query together with all the meta information to a XML document as shown in listing 2.2. Unlike in the example, it is common to use the service module class ID as handler name.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<jxta:ResolverQuery xmlns:jxta="http://jxta.org">
  <SrcPeerID>urn:jxta:uuid-59616261646162614E50472050325033\
    1A227980E5924E80A3FD8ECD73D4C31803</SrcPeerID>
  <HandlerName>BeerFinder</HandlerName>
  <QueryID>1</QueryID>
  <HC>0</HC>
  <Query>Got a beer?</Query>
  <SrcPeerRoute><jxta:RA><Dst>
    <jxta:APA><EA>tcp://129.132.105.170:32725</EA></jxta:APA>
    </Dst></jxta:RA></SrcPeerRoute>
</jxta:ResolverQuery>
```

Listing 2.2: A sample resolver query XML document

The hop count, which is incremented on each hop, ensures that the query is not sent to far away. Although the rendezvous service already does such a check, we have to do it here again because the resolver does not always need to use the rendezvous propagation mechanism.

Queries and responses may be propagated or sent directly to a specified peer. The resolver service is not necessarily dependent on the rendezvous service but would of course be limited to local network propagation and single message dispatching in a situation without rendezvous.

A resolver response looks basically the same as a query. The actual response is also a string. There is no hop count as it does not make any sense here. Also the source peer ID and route are dropped but there is a response peer ID, so the recipient knows from which peer the response originates.

A client service sending a query should be able to register a listener for related responses, so it does not have to check itself if the response matches the query. That is one of the main tasks of the resolver service.

[16]The service name is a little confusing: it means actually to resolve queries to responses…

2.4.5 Discovery service

Whereas the resolver does not maintain much data and only serves as an intermediate message layer for other services, the *discovery service* is a fundamental part in JXTA and much information passes it.

The discovery service is dealing with all sorts of advertisements, so it knows about all peer resources (as advertisements are promoting resources). On one hand, it serves as an advertisement storage and on the other hand, it is responsible for finding remote advertisements and for letting other peers know about the locally stored ones.

Discovery queries and responses Whenever a service needs an advertisement, it does a local discovery query which is equivalent to a storage lookup.

Generally, we distinguish between peer, peer group and other advertisements, so we specify the advertisement type in a query. A query may restrict the search additionally with a key and a value name. The key is a XML tag name. It is allowed for the value to contain the wild char `*` in the beginning and/or at the end. The number of answers may also be limited by setting a threshold.

Querying remote peers is actually the same but one may choose to send a query to a single peer or to propagate the message in the group. In both cases, the resolver service is actually used for sending the messages.

The discovery query is an XML document specifying query type, key, value, threshold and optionally also the source peer advertisement. Looking at the final endpoint message, we see a resolver element containing a resolver query XML document which contains the quoted discovery query XML data[17].

A peer is not obliged to respond to any remote discovery query. Peers which have sent a query should expect no, one or multiple responses. They cannot expect that the threshold is respected, neither as minimum nor maximum.

A discovery response message may contain several matching advertisements. It may additionally also contain the responding peer advertisement.

Discovery responses are not only used to respond to queries. It is also allowed to publish advertisements to other peers, especially the rendezvous server, using a "response" message.

To feed the local storage with advertisements, one just publishes them locally.

Shared Resource Distributed Index As we have already seen, a rendezvous server will not propagate every message. In the case of discovery queries, the rendezvous makes use of a *shared resource distributed index*, henceforth called SRDI (see [Pro06]).

The SRDI is an advertisement index containing certain keys and values together with peer IDs enabling the rendezvous to lead queries to peers which actually should have matching advertisements. This monumentally reduces network traffic as peers that do not have the needed information are not queried.

[17]The discovery query is quoted because the resolver service currently expects a simple query string.

Note that the SRDI does not contain the entire advertisements but only has some important keys and their values for every advertisement a peer has.

But how does the rendezvous maintain its SRDI? Every edge peer sends its SRDI to its rendezvous. When newly joining the group, it sends the full index. Later, it sends regularly (for example every minute) a SRDI delta, that means only the key/values for newly discovered, created or updated advertisements. When a rendezvous lease is canceled, the peer's SRDI entries are removed automatically by the rendezvous.

There exists a generic SRDI XML document used for pushing SRDI entries to the rendezvous. It contains also a TTL, so an SRDI entry is not valid for ever (like the advertisements themselves).

The current JXTA implementations do not index each advertisement's XML tag. When creating a new advertisement type, one should specify which elements are important and should therefore be indexed. When speaking in database terms, one should at least index the primary key attributes.

2.5 JXTA's P2P infrastructure and peer roles

To bring some clarification into the partly insufficiently introduced peer roles, we would now like to show now a short overview of the entire JXTA P2P infrastructure:

Although JXTA may use central rendezvous server lists when starting up, we can definitely see JXTA's structure as a *true* P2P system. It does not rely on central servers for any core task and uses the P2P structure for all purposes.

However, there are various peer roles in the network although there may also exist combinations:

Edge peer Whenever traffic or CPU power is expensive, an *edge peer* is surely the right role for a peer. Such a peer heavily relies on other peers and consumes parts of their attention. Most peers would actually choose this kind of role.

Rendezvous peer A *rendezvous peer* is providing a rendezvous server and enables edge peers to make contact with other peers. In a JXTA network, we need at least one rendezvous server because we usually want to discover other peers and do not have the physical locations hard coded of other peers that we want to communicate with.

Router peer A *router peer* enables peers to communicate with others to which they cannot connect directly. This is used for peers behind a NAT gateway or a firewall. Therefore, router peers may have to manage all their clients message traffic in one or both directions.

Note that peer roles may dynamically change. For example, a peer which cannot find any rendezvous server could automatically become a rendezvous. This is of course adjustable in the platform configuration.

3 Design

Keeping our goal in mind, we focus on the essential parts needed for Origo (see the requirements in section 3.1) because we cannot port *all* JXTA protocols to Eiffel within the given six months of this thesis. See chapter 8.1 for future work and how *VamPeer* may be extended to add missing features.

3.1 Requirements

As we are implementing JXTA for Eiffel with regard to support Origo's network layer, we state the following demands:

1. The Origo peers may communicate among each other without being disturbed by messages from other peer applications.
2. An Origo peer is on the one hand able to advertise its existence to the peer group and on the other hand to discover other peers, especially a "core" peer.
3. An Origo peer is able to send messages to other peers. A message may contain data of any type and length.

Mapping these ideas to the JXTA world, we specify the following requirements:

To fulfill the first demand, we should be able to support private peer groups. This means also that we need to be able to run an own JXTA infrastructure without using foreign resources on the net. This is exactly what JXTA teaches us to do for peer applications.

To enable peer discovery in JXTA (second demand), we need a set of services: Obviously, we need at least the *discovery service* which allows us to publish and query for (peer) advertisements. Then, we need the *resolver service* which the discovery depends on[1].

But to get into contact with unknown peers, we heavily rely on the *rendezvous service*. The rendezvous server is the first peer we contact and the advantages of the discovery service only is possible with the rendezvous' help.

Whereas a rendezvous server is needed for the entire peer group, not every peer needs to implement the server part. An Origo peer may be a rendezvous client only. Therefore, we concentrate on the client part and note the possibility to run the rendezvous server as a JXTA JSE peer.

[1]See subsection 2.4 for a short introduction to the mentioned services.

As all discovery messages (and also messages from the resolver and rendezvous) are based on normal JXTA messages, we clearly need the *endpoint service* together with a transport module. Having these, we honor also the third demand requesting a message transport.

There is still a missing service: the *endpoint router*. Peers are addressed with peer IDs (see subsection 2.2) so we need the endpoint router to resolve the IDs to addresses that specify the transport protocol and the exact address. This is only a small task of the router. There is no urgent need for the other functionality enabling us to have peers behind firewalls and NAT gateways.

Summing up the set of required services, we get the list shown in table 3.1. The requirements are fairly vague but we will enlighten the details later in this section and show the resulting challenge when presenting the implementation in chapter 4.

Service module	*Functionality*
Endpoint service	Message layer abstraction
A transport module (e.g. TCP)	Message sending and receiving over the wire
Endpoint router (parts)	Routing messages to available gateways selecting a fast transport
Rendezvous service (client)	Connection to peer group
Resolver service	Query-response system
Discovery service	Advertisement querying and publishing

Table 3.1: Required services

We now move on to the *VamPeer*'s design. By first introducing the module structure, we see how the entire platform works.

3.2 Module structure

The JXTA structure is very modular; every service and every peer group, even the platform (the world peer group) itself, is a module.

A module is an entity which can be started, suspended and stopped. This enables the *VamPeer* platform to perform the entire start up process without knowing every modules' internal details.

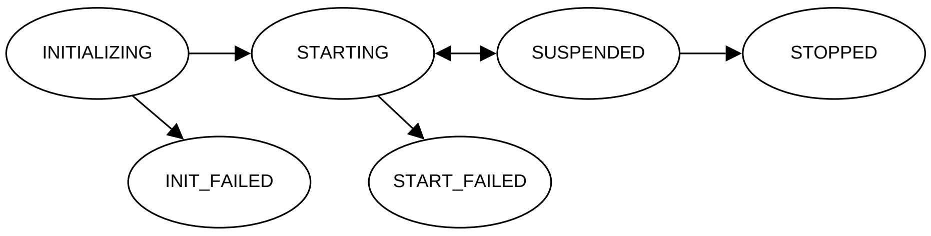

Figure 3.1: Module life cycle

Figure 3.1 shows a module's life cycle. After successful loading where usually the basic initialization like creating data structures is done, a module can be started. The `suspended` mode is available to temporarily stop a service in order to make it rest for a while in standby. The `start` method has to take care of the two possible calling states. To permanently shutdown a module, one can call `stop` in suspended mode. A stopped module should not and cannot be started again. If you really need to do this, you have to create a new module instance.

There are also some states indicating fatal errors. When one of them has occurred, a module should not be touched again. Only the constructive operations `init` and `start` are allowed to produce errors. `suspend` and `stop` are always expected to function properly.

A module is represented in *VamPeer* with the deferred class `P2P_MODULE`. Figure 3.2 gives an idea about the classes which effect it[2]. Also, we already see how the peer groups are related with modules; that is what we will look at in the next subsection.

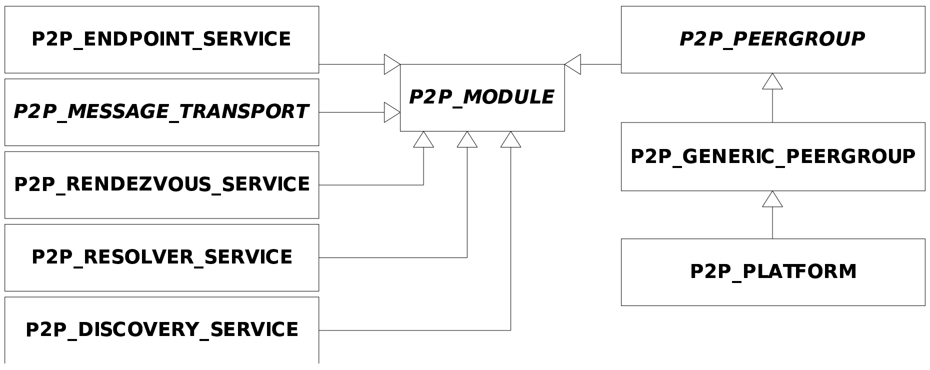

Figure 3.2: Module class hierarchy

3.2.1 Peer group modules

As a peer group specifies the available services for its group, it makes sense to make a peer group responsible for managing its services. So we just have to start the peer group when we would like to start the application's P2P support with all services. Therefore, the peer group is also a module which can be started and stopped (`P2P_PEERGROUP`).

Unfortunately, it is not that easy to start the entire P2P platform. To load a module we need a parent peer group, an ID and a module implementation advertisement[3] used for configuration. Thus, we need a bootstrapping process that handles the loading of our main peer group—usually the net peer group.

This is exactly the purpose of our world peer group (`P2P_PLATFORM`). In *VamPeer*, it is not used for anything else but loading the NPG. As the WPG is itself a peer

[2]A deferred class in Eiffel is like an abstract class in Java; to effect such a class means to implement deferred features (abstract methods).

[3]See sections 2.2 and 2.3.

group, it is also a module but with a different creation procedure. This allows us to retrieve the needed data like the configuration directory path and the logger object.

It is not yet clear enough why we do not just adapt the net peer group to manage the platform creation. At the moment, only one real peer group can be run in *VamPeer* but this could more or less easily be changed. By then, generic peer groups (represented through `P2P_GENERIC_PEERGROUP`) should not be involved in the bootstrapping process. See section 3.3 for more details about how peer groups are defined in *VamPeer*.

In JXTA JSE, the entire P2P application is also a module controlled by the JXTA's module loader. We do not go so far in *VamPeer*: The P2P environment should only be a part of the user's application and should not dominate everything.

3.3 Defining a peer group

Defining a peer group implies to prepare several requirements:

As a peer group is in the first place a normal module, we first have to establish the module configuration. This requires to have a module class ID (MCID) and a module specification ID (MSID) which identify the local and remote behavior as already pointed out in section 2.4.

Another part of the group definition is built by the group ID, a name and a description. Together with the MSID, we are now able to build the peer group advertisement.

Modules are loaded by the MSID. This means that the module loader gets a MSID (besides an arbitrary ID and a name) in order to load the correct module code. Hence when loading a peer group, we have to provide its MSID.

As peer groups define which services (modules) they provide, the peer group module code is responsible to load these modules. The group module therefore somehow contains a list of needed module MSIDs. In our standard group implementation `P2P_GENERIC_PEERGROUP`, this list is called `modules` and `define_modules` is the method that initializes it. As we use a parent-child relationship between groups, the group services are usually inherited by a child group but one may easily redefine them.

When we inherit group services, we share the module instance. This clearly makes sense for certain services which are not group context sensitive. The top group which defines such a service is responsible for it (it is the only authorized group to load, start and stop this module). To respect this rule, the group's module list also has to keep track of whether a module is inherited or newly defined.

When a module is loaded, the loader and later also the module itself should be able to access the module implementation advertisement. The loader may need the advertisement to know what module code to load for the given MSID. The module itself may use the advertisement to lookup some configuration parameters. Thus, the module implementation advertisement has to be available for each used module.

Conventionally, the group module implementation advertisement contains all advertisement of its modules. The peer group module will then extract these and make them singly available.

There is one open issue with module loading: a running module has itself an ID. We speak thereof of the *assigned ID* because the module loader assigns an arbitrary ID to the module. Usually, we assign the MCID but it is not necessary to do so. The module uses its ID to create a unique handler name when registering with other services.

Summarizing, we need for each module an implementation advertisement where the one for the group contains all advertisements for its services. Additionally, we should provide a peer group advertisement to declare the group module as a JXTA peer group.

The entire module loading procedure is quite generic because of the dynamic module loading. Although Eiffel does not provide this, we choose to stick to the convention and provide also these implementation advertisements even if we only may use it to parametrize modules.

Unhappily, we need code in Eiffel where we specify which classes have to be loaded for a given MSID. To create a peer group with user services, we therefore have to heir from `P2P_GENERIC_PEERGROUP`, adapt the modules list and redefine `load_extern_module` to be able to specify the Eiffel class which should be used. To load a private net peer group, the platform `P2P_PLATFORM` provides the possibility to specify a net group loader in terms of an agent.

3.4 Services

With the strict module structure, we are basically done with presenting the *VamPeer*'s design because everything is bundled into a module and therefore every service looks quite similar. Nevertheless, we have to describe some particularities, especially how services interact among each other.

3.4.1 Module choice

Each module resides in its own Eiffel cluster together with its related classes, namely the XML document types. Module unspecific classes are located in the main `vampeer` cluster. See figure 3.1 for a clusters overview.

```
vampeer/            (38 classes)
|-- discovery/      ( 3 classes)
|-- endpoint/       ( 4 classes)
|-- pipe/           ( 2 classes)
|-- rendezvous/     ( 5 classes)
|-- resolver/       ( 5 classes)
`-- transports/     ( 7 classes)
    |-- router/     ( 4 classes)
    `-- tcp/        ( 5 classes)
```

Listing 3.1: Clusters overview

The pipe service[4] is not actually implemented but the module specification advertisement depends on the pipe advertisement.

Until now, we have mostly spoken of generic transport modules but finally we implement only one, the TCP transport.

TCP is actually a dumb name as most of the other transports indirectly are TCP based too. But with the TCP transport, we directly lie on top of TCP using the JXTA's wire representation for messages. It is the most simple but also the fastest transport, which explains our choice.

3.4.2 Service layers

We would like to clarify how all the JXTA services are related to each other. Particularly, how messages are passed through. We do so by looking at two examples which cover all services implemented in *VamPeer*. We first treat an outgoing and later an incoming message.

Outgoing message Lets look at a discovery query which is sent out to be propagated in the group. It could be a general query to find new peers. Figure 3.3 shows the UML sequence diagram hiding the exact operation signatures. We will comment each method call:

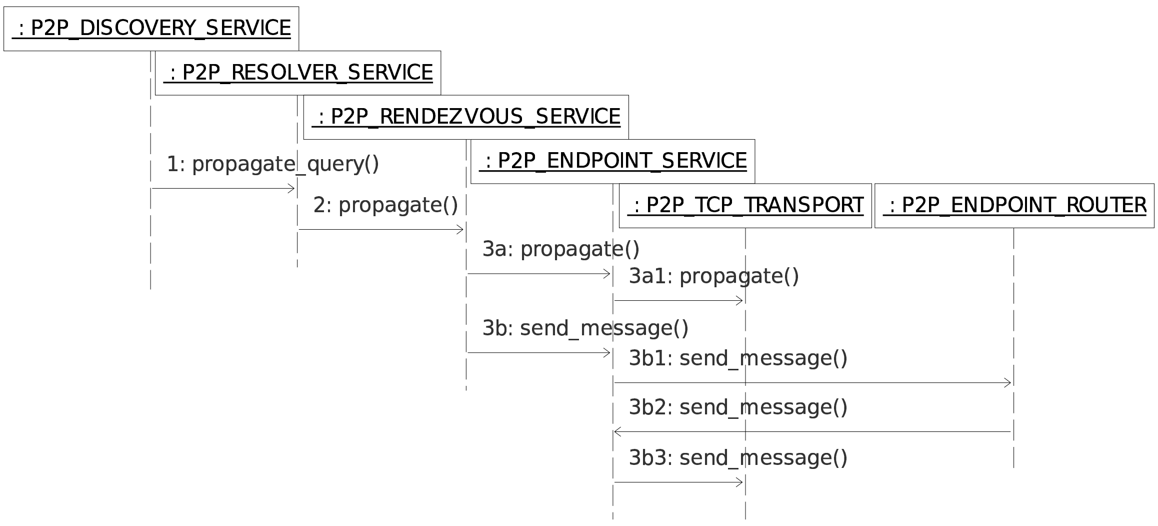

Figure 3.3: Information flow for an outgoing discovery query

1 When the discovery service is called with `remote_query_advertisements`, it creates a `P2P_RESOLVER_QUERY` containing the discovery query string and a handler name (see the incoming message example for its use). It then passes it to

[4]The *pipe service* implements the pipe binding protocol and provides virtual communication channels among several peers.

the resolver which is requested to propagate the query (instead of just sending the message to a single peer).

2 The resolver service then creates a P2P_ENDPOINT_MESSAGE with a message element containing the resolver query string. It passes the message together with the resolver's service name to the rendezvous service.

3a The rendezvous adds another message element with some meta data to the message. This informs the recipients that the message was propagated (important when they do repropagation). The rendezvous then first propagates the message in the local network by simply calling the endpoint's propagation service.

3a1 The endpoint service now passes the message and the service name to each transport's propagation method. Actually, it makes only sense for the TCP transport, as it supports IP multicast (other transports just ignore the call). However, multicast is discouraged[5] and mostly turned off in the platform configuration.

3b When the peer is connected to a rendezvous server, it is able to do propagation via this server. So the rendezvous service sends the message to the server's peer ID by calling the endpoint service's `send_message`. For this, it has to create a P2P_ENDPOINT_ADDRESS with the `jxta` protocol and the server's peer ID[6].

3b1 To resolve the endpoint address, the endpoint service passes the message to the endpoint router which is the registered transport module for the `jxta` protocol.

3b2 The endpoint router does a (local) lookup for the given destination peer ID (querying for peer and route advertisements). As soon as a gateway and a transport protocol is chosen, it calls the endpoint service again to deliver the message with a rewritten, specific destination address[7].

3b3 The endpoint service now passes the message to the specified transport protocol which tries to connect to the specified peer and writes the message to the wire. Note that there is neither a feedback to the caller whether the message dispatching has been successful or not, nor an acknowledgment message from the other peer. The TCP transport is therefore seen as an unreliable transport.

Incoming message The path for incoming messages is somewhat shorter. We will continue our example and let the peer receive a discovery response. To explain the three method calls, we first look at figure 3.4 which presents the calling sequence.

[5]Multicasting should not be used because it causes much network traffic and may stress some smaller edge peers. It poses also a risk for developers that test in local networks only because things may work locally with multicast but maybe will not with remote peers.

[6]E.g. `jxta://rdv-peer-id/rsv-service-name` (while rsv-service-name is the resolver service module class ID).

[7]E.g. `tcp://129.132.105.170:9700/rsv-service-name`

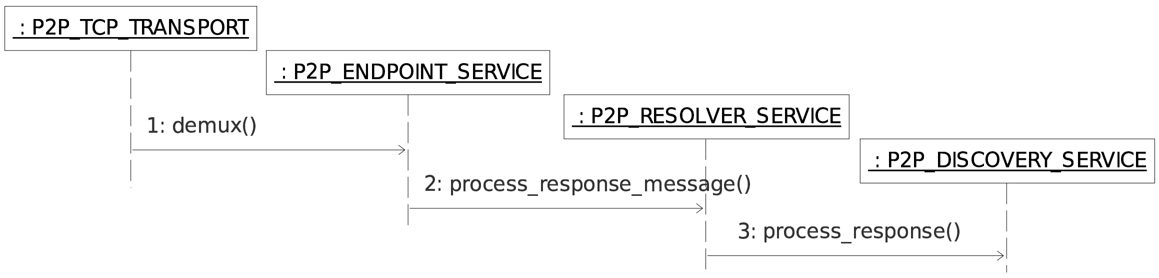

Figure 3.4: Information flow for an incoming discovery response

1 As soon as the message transport has received a message and the message parsing from the wire has been successful, it calls the endpoint service's `demux` method with a `P2P_ENDPOINT_MESSAGE` object.

2 From the delivered message, the endpoint service extracts the destination address and therefrom the service name. Then it calls the registered agent for this server name which is owned in our case by the resolver service.

3 The resolver interprets the resolver message element and is able to extract the response string and a handler name. It then calls the registered agent for this handler name and passes a `P2P_RESOLVER_RESPONSE` object.

  The discovery service will finally parse the resolver response string and create a `P2P_DISCOVERY_RESPONSE`. From there, the delivered advertisements (responses) are either published locally or they may be passed to a further agent registered by a user service.

3.5 Address rewriting

To stay compatible with other JXTA implementations, we have to pay attention to a special and a little cumbersome topic: *address rewriting* (address mangling). The problem is that messages may not be directed to the correct service in other implementations since they may have a slightly other peer group hierarchy.

In a JXTA platform, only one module can actually handle network traffic (one module per transport protocol). But generally, it is possible to run multiple peer groups at a peer such that each group has its own endpoint service. Hence, the question is where we register the transport module(s).

As the JXTA protocol does not specify this, each implementation can do as it likes. While in JXTA JSE the WPG owns the transport modules, JXTA-C and *VamPeer* settle them in the NPG. The reason for that is that in Java, the WPG is a real peer group whereas JXTA-C does not have a WPG and we *VamPeer* guys only use it for platform configuration purposes.

With multiple endpoint services and in order to receive messages, services have also to be registered in the top peer group that owns the transport modules. Such an indirect registration, which is automatically done behind the scenes, is done with a *mangled* address which includes the original peer group ID where the service is registered in the first point. Figure 3.5 shows how such a mangled address is assembled.

$$\underbrace{\underbrace{\texttt{EndpointService:}}_{\text{Mangling prefix}}\ \underbrace{\texttt{jxta-NetGroup}}_{\text{Service's group id}}}_{\text{New service name}}\ /\,\underbrace{\underbrace{\texttt{service\_name/service\_parameter}}_{\text{Original service name/parameter}}}_{\text{New service parameter}}$$

Figure 3.5: A mangled service handler name

With such a mangling scheme, it is clear that also the message destination address has to obey this rule. To send a NPG discovery message to a Java peer, we have to mangle the address before because the discovery is in the NPG but the Java's transports are in the WPG. Sending the same message to a *VamPeer* peer, we would not have to mangle the address because its transport modules are also in the NPG.

That is really bad, but it gets worse when we look at the reverse direction: A Java peer will send discovery messages always mangled and we would discard the message because we have no mangled address registered for it as our discovery service already runs in the top group NPG.

To overcome these troubles, we always register our services also with the mangled address in order to not loose any messages from Java peers. And to send messages, we may always need to mangle the destination addresses but we do not force it; hence, it is still possible to speak with other *VamPeer* peers without group mangling.

3.6 Rendezvous propagation

In P2P networks, propagation is a central service because for many messages we do not know which peer exactly may use the information. So, we just propagate the message to everyone and hope that some peers may use or process it. As we have already seen, propagation with group scope (instead of local network only) is done by the rendezvous service.

When we hear about message propagation, we might be tempted to classify it as flooding the network. While this may be a solution (actually, previous versions of JXTA JSE did this), it is rather traffic consuming and extremely inefficient.

We already mentioned in section 2.4.3 that the rendezvous today just passes the messages to the appropriate services which decide based on some gathered knowledge if the message should be repropagated or not.

But this means that we have to get away from the idea that we may only place a JSE rendezvous somewhere and propagation just works. Either, we design our peer application to use only standard services or we implement the user services also in Java on the rendezvous peer. The third and best solution would be of course to have an Eiffel rendezvous implementation but this is not possible due to lack of time.

4 Implementation

In this chapter, we go into the implementation matter and describe solutions for the most important tasks. The library code is rather described in the chapter 5 showing how we should use the *VamPeer* library.

4.1 Development environment

During the design step, UML class and sequence diagrams were used by means of the open source tool *Umbrello* [Hen06], which laudably supports diagram exports to a vector-based graphic format, namely EPS.

During development, we used *EiffelStudio 5.7* [Eif06b] and therefore the ISE compiler. The new introduced configuration file format ECF would be a mercy but it is quite cumbersome to integrate older libraries because ECF is (currently) not widely accepted.

As we stumbled around a lot with networking stuff, we used *Ethereal* [Com06] (now renamed to *Wireshark*) to sniff packets sent on the network.

For testing, we used the standard Gobo tool *getest*, which provides a very easy and comfortable way to write unit tests.

4.2 Used libraries

VamPeer is of course using several software libraries. For the first, it is obvious that we use *Gobo* [Bez07], in the current release 3.5. We need it for data structures, date/time and its XML generation and parsing support.

As we are mainly dealing with remote peers, we need a networking library. We looked for a simple solution and got to the *EiffelNet* code [Eif06a]. Unfortunately, the library is not used by many people resulting in an API which is not always useful for each task. Therefore, we wrote an extension to it, see subsection 4.3.

To support various independent tasks at the same time, we use the *EiffelThread* library [Eif06c] and therefore switch the `multithreaded` flag on.

For the logging task, we make use of the great *Log4E* tool [Goa07] which is now part of the *Goanna* project. It provides a synchronous logging mechanism similar to Jakarta's well-known Log4J.

We provide ECF and XAce files for all used libraries in the `contrib/` directory.

4.3 Socket extensions

The main problem we have with *EiffelNet* is the missing support for timeouts. Whenever we wait for network data, we do not like to wait eternally. Generally, there are two kinds of interrupts in these situations in which we like to quit the reading/waiting task: first, when a certain time has been passed and second, when we get an internal request for closing the connection such when the user application is shutting down.

Another lack in *EiffelNet* is buffering. Usually, we like to assemble a network message and send it as *one* packet. The straightforward idea to just use a string does not really work well because with binary data, we have to append 8-, 16- or 32-bit integers (respecting the network byte order!)—this results in code which we certainly never like to see here and there in our application. Additionally, we sometimes have to know how large a buffer is (for example to specify a message body length).

We therefore wrote a helper class `P2P_SOCKET_EXTENSIONS` which provides exactly these features: timing and buffering. We did not choose to create a socket heir class because the helper methods may be used for several types of sockets: TCP and UDP. We provide methods for writing by using a buffer, for reading and for some socket checks.

One may fill the buffer with strings and integers (from 8 to 64 bits, using big-endian format). The buffer is just a string and may be adapted and used at will. As soon as the buffer is sent, the buffer is emptied again.

For the read methods (returning a string or the various integer types), one has to set a timeout first. They read and wait until either they got exactly the expected amount of data, they pass the timeout or a given constraint has become active. The last two cases raise an exception. Of course for strings, there is also a method reading *up to* a given data length (in many cases, we do not know exactly how much data we are expecting).

The socket checks provide a method to wait until a connect request is successful within the timeout and a method to check generally whether data is available for reading.

The timeouts are implemented with the socket's non-blocking mode. So, the socket's read command instantly raises an error when not enough data is available. We catch it and put our thread to sleep but regularly check for new data. When we have slept too long, we raise an exception to the user.

Because of the non-blocking mode, we get a lot of exceptions during a *VamPeer*'s run. So for development, we should disable exception handling for the following exception types: 21 (I/O exception), 23 (Retrieve exception), 24 (Developer exception) and 27 (Runtime I/O exception). Therewith, we are not bothered with the network latencies.

The mentioned constraint is an agent that can be defined and which is called every time before a thread waits. When the agent returns `True`, the actual task is interrupted as described above.

Creating this socket extensions class, we have mastered a lot of networking troubles at a single point.

4.4 XML documents

JXTA makes heavy use of XML documents. Not only every advertisement is in XML, but also every higher level JXTA message. Therefore, we need a simple way to parse and generate XML documents.

Figure 4.1 shows our class structure for XML documents. We hide the exact operation signatures for simplicity and do not list all descendant classes.

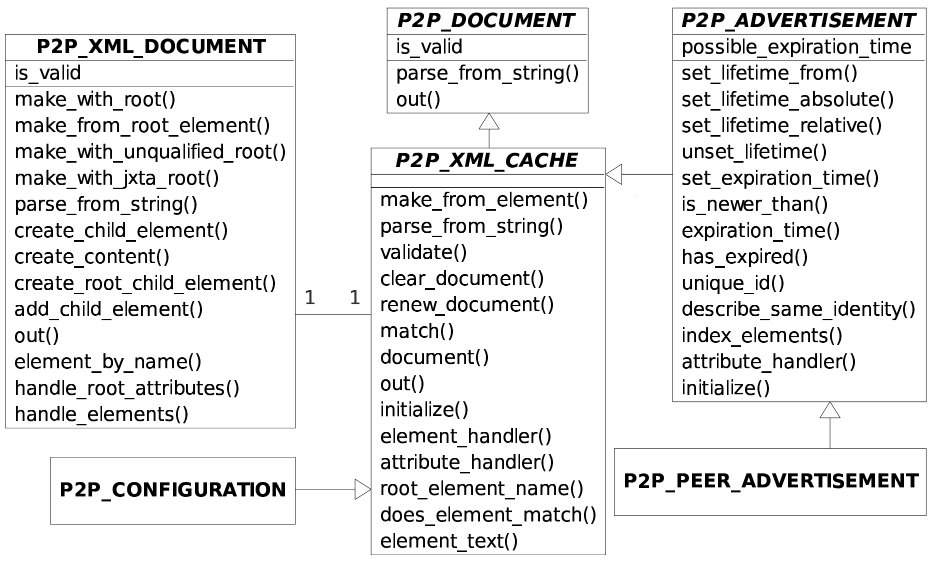

Figure 4.1: XML document class hierarchy

The class P2P_XML_DOCUMENT manages most handling with the *Gobo XML* interface. It is not an abstraction so that the underlying XML library could easily be replaced; it rather provides helper methods just to simply build an XML tree from elements, to get the content as a string and to build a tree parsed out of a string. When parsing, we let Gobo build the full tree and we afterwards provide callbacks for each root element and root attribute.

Its main client class is P2P_XML_CACHE (deferred) where all XML document classes are inheriting from. It mainly declares central document methods, such as creation, validation, output and element matching. It inherits from P2P_DOCUMENT which provides an interface for very generic documents. The XML tree is not built until `out` is called the first time. Further calls return a cached XML string unless any element has been changed (`renew_document` should always be called internally in element setters).

To create a new XML document type, we just need to inherit from P2P_XML_-CACHE, define some setters/getters for the actual content elements and implement `match`, `root_element_name`, `attribute_handler`, `element_handler` (to gather data after the parsing process) and `document` (to create the XML elements). We may also redefine `initialize` and `validate` and add additional creation methods.

Whereas this interface is appropriate for general XML documents, we require an additional interface for advertisements: P2P_ADVERTISEMENT. Each advertisement has a unique ID used for the advertisements store in the discovery service. It should also define its lifetime and the remote expiration time. For SRDI, an advertisement should also define some elements which can be indexed.

Hence, we declare a unified interface for advertisements as shown in figure 4.1. All advertisement classes (such as `P2P_PEER_ADVERTISEMENT`) should effect it.

4.5 Using UUID for JXTA IDs

For *VamPeer*, we use JXTA IDs in the UUID format as it is the case in JXTA JSE. Look at figure 4.2 to get an overview of the ID class hierarchy.

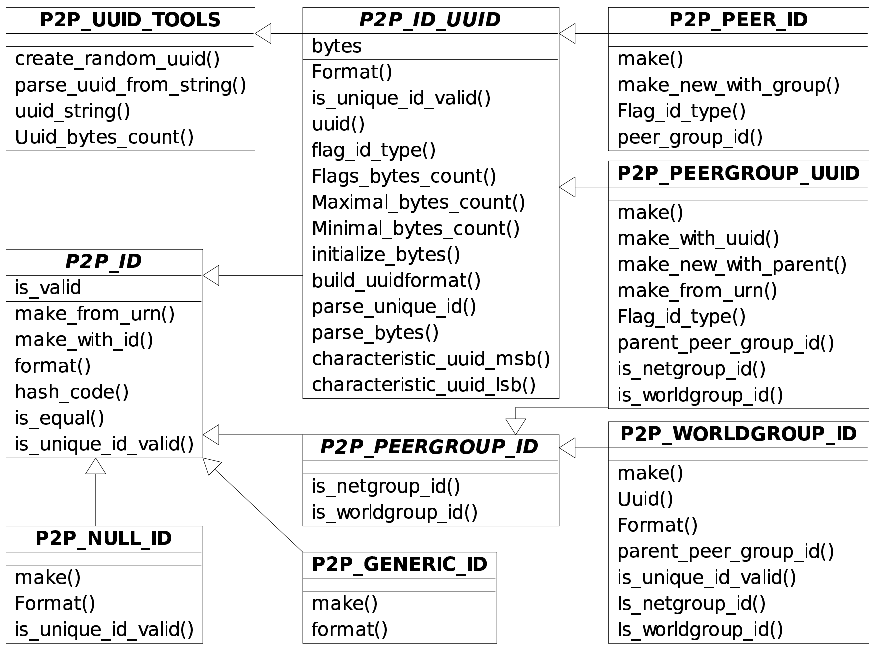

Figure 4.2: ID class hierarchy

`P2P_ID` is the main interface for an ID. Such an ID can be parsed out of and written as a URN string. It is comparable and hashable. Direct heirs that can be instantiated are `P2P_NULL_ID` (for referencing no resource, actually never used in *VamPeer*) and `P2P_GENERIC_ID` which may contain any valid ID in any format.

We support the `uuid` format using the deferred class `P2P_ID_UUID`. It parses the unique ID part and creates a byte array. For each ID type (such as peer, peer group, codat ID...), there is an effecting class defining the byte interpretation. Figure 4.2 evinces only two of them. Each such type also defines the characteristic UUID bytes, made available through the feature `uuid`[1].

[1]While the peer group UUID in a peer ID is just meta information, the peer UUID is the characteristic part.

`P2P_UUID_TOOLS` is actually not related to the ID class structure but provides helper methods to create a random UUID and to parse/write a UUID. In *VamPeer*, UUIDs are everywhere seen as `ARRAY [NATURAL_8]` with a capacity of 16 bytes.

For peer group IDs, there is a special interface `P2P_PEERGROUP_ID`. It is independent on the `uuid` since we use other formats as well for peer group IDs, for example the `jxta` format for the net and the world peer group. The `uuid` version `P2P_PEERGROUP_UUID` is therefore an heir of two classes, `P2P_PEERGROUP_ID` and `P2P_ID_UUID`.

It may be confusing that the `jxta` peer group IDs also provide a `uuid` feature. It is used to generate child peer group IDs because children point to the parent group uuid. The following examples try to clarify these ID relationships:

World peer group ID: `urn:jxta:jxta-WorldGroup`
Implicit UUID: `59616261646162614A78746150325033`[2]

Private net peer group ID:

$$\texttt{urn:jxta:uuid-}\underbrace{\texttt{822A7C9E6B804759870B81B10070E9C9}}_{\text{Group UUID}}\underbrace{\texttt{59616261646162614A78746150325033}}_{\text{WPG's characteristic UUID}}\texttt{02}$$

Peer ID, member of the private net peer group:

$$\texttt{urn:jxta:uuid-}\underbrace{\texttt{822A7C9E6B804759870B81B10070E9C9}}_{\text{Group's characteristic UUID}}\underbrace{\texttt{2F3F01367359485B95D5C6CFA82B9775}}_{\text{Peer UUID}}\texttt{03}$$

4.6 Threads

We spend an own section to the thread handling in *VamPeer* because we would like to present which threads exist and because users have to pay attention to some details.

Threads are an absolute must when dealing with the network. But threads are also a source for many software bugs because we cannot rely on all contracts anymore in a multithreaded environment. Other threads may change something in the course of time. This is why we have to use locking mechanisms to lock other threads out for a certain amount of time.

While we have `synchronized` environments in Java, Eiffel currently does not support such a mechanism in the language itself[3]. We therefore have to use the possibilities from *EiffelThread* which provides read/write locks (for multiple readers, one writer), mutexes (simple locking) and condition variables (for thread synchronization).

To gain a solid peer application, we had to identify the shared objects between the various threads and to introduce appropriate locks. We wanted our application also

[2]In ASCII, the 16 bytes stand for "YabadabaJxtaP2P3". Actually, the last byte was an error; it should be a "!" but they looked up the wrong number in the ASCII table (decimal instead of hex)...

[3]SCOOP will be a good solution but it is still a research topic and the current version not stable enough.

to terminate correctly which furthermore means to appropriately destroy a lock at a single place. This took us a considerable amount of time…

In *VamPeer*, we are using at least four threads besides the main thread which we will all describe now.

Main thread First, we will look at what the main thread is used for. It is the thread which starts the platform and is in full control by the user application.

Thus, the entire platform starting is done and all the other threads are created by the main thread. Since *VamPeer* does not provide an event loop, the user application has to handle its main thread itself.

Usually, the main thread is also used to shutdown the platform.

4.6.1 TCP Transport

Most of the threads are created by the TCP transport and its related classes.

Server thread The first thread created is the server thread. `P2P_TCP_SERVER` binds a server port and listens for incoming connections. When a new connection is accepted, it passes the socket to the main tcp class (`P2P_TCP_TRANSPORT`) which will handle the connection with another thread.

The server thread can be closed by calling `shutdown` (from another thread). The socket will be cleaned up which causes the server thread to terminate instantly.

Connection threads Each tcp connection gets its own thread. As soon as a socket is passed from an incoming connection, or when a new socket has to be created, the connection object `P2P_TCP_CONNECTION` launches a new thread.

The thread manages waiting during connection setup and waiting for incoming messages via its socket. It is important to know that message processing (for received messages only) is handled by the connection thread. This means, a service should never do extensive work in a processing agent.

It is therefore not possible for a message handler to wait for another message retrieved with the same connection. However, sending messages is allowed in such a handler (see next paragraph).

A connection thread terminates itself after a certain time but can of course also be destroyed by calling `close`.

Message queue thread Because services should not be affected by the slow message delivery, outgoing messages are queued. The calling service thread returns instantly and without result as soon as the message has been successfully stored in our message queue.

`P2P_TCP_TRANSPORT` manages a message queue thread which waits for and gets triggered on new messages. It then looks up a possible existing connection or tries to open a new one. As soon as a connection state is in a valid mode, it sends the message and removes it from the queue. It tries several times when the dispatching fails.

Summarizing, we have two fix threads for the TCP transport plus a thread per connection.

In suspend mode, all connections are closed and the server as well as the queue manager thread are stopped.

4.6.2 Rendezvous connection manager

The rendezvous service owns another thread. It is only used to stay in contact with rendezvous servers meaning to renew connection leases. Hence, this thread is mostly sleeping.

The thread is launched when the service is started and is stopped when the service gets suspended. During start, we have a bootstrapping problem because not all services are started yet and the system may only partially be functional. We cannot fire off sending messages in this state and have to wait until the platform is started entirely.

That is the reason why the peer group is offering a method `when_fully_started` returning `True` as soon as all group services are started successfully or `False` on a failure. The rendezvous connection manager thread therefore calls this method before it begins to contact a rendezvous server to get a lease.

4.6.3 Discovery SRDI

The discovery service creates a thread to manage SRDI pushes. It regularly looks for new local advertisements and propagates its index changes.

Like the rendezvous connection manager, it waits with index pushing until the platform has been started fully. The thread is stopped also in suspend mode.

4.7 Advertisement storage

Probably one of the most important part in the entire platform is the advertisement storage. Services and user applications should not have to store advertisements itself but should be able to access a central storage.

That is what the discovery service provides with its local query methods. We would like to describe its advertisement storage here. The storage can be divided into two parts, persistent and memory storage.

4.7.1 Persistent storage

When starting the platform, a configuration directory must be declared. This is used to store the platform configuration and some advertisements.

```
|-- Modules/
|   |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010206.xml
|   |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000020106.xml
|   |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000030106.xml
|   |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000060106.xml
|   |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000080106.xml
|   |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000090106.xml
|   `-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000B0106.xml
|-- jxta:jxta-NetGroup/
|   |-- Advs/
|   |-- Peers/
|   |   `--jxta:uuid-59616261646162614E504...3D4C31803.xml
|   `-- PeerGroup.xml
|-- jxta:jxta-WorldGroup/
|   |-- Advs/
|   |-- Peers/
|   |   `--jxta:uuid-59616261646162614E504...3D4C31803.xml
|   `-- PeerGroup.xml
`-- PlatformConfig.xml
```

Listing 4.1: Persistent storage directory layout

The file hierarchy layout is very simple as shown in listing 4.1. Mainly, for each peer group there exists a directory containing its group advertisement, one directory for general and one for the peer advertisements. Each advertisement is stored as a single XML file with its shortened unique ID as file name. For the module implementation advertisements, a special modules directory is maintained.

The persistent cache is only used when the platform is loaded. During the initialization process, configuration will be read from disk or it will be created and stored. When the configuration directory is not present, it will be created.

All the advertisements will be loaded into memory once the discovery service has been started. This enables the peer application to have advertisements permanently available.

The class P2P_CACHE_MANAGER is responsible for accessing and managing the persistent cache. As soon as a platform instance is disposable, the cache manager is ready for access too.

However, the cache currently suffers from a drawback: It cannot handle advertisement lifetimes and expiration times. A stored advertisement will never be discarded because the lifetime is not part of the XML document itself and therefore cannot be stored to disk. That is the reason why new advertisement are never written back to the cache. Of course, the peer application may itself write some chosen advertisements to disk using the cache manager.

The persistent cache is thus only used for really permanent advertisements. By default, these are the module implementation, the peer group and the own peer advertisement(s).

4.7.2 LRU cache

The memory cache is implemented with a *Least Recently Used* (LRU) cache in the class `P2P_ADVERTISEMENTS_LRUCACHE`. It is inspired by the JXTA JXME project which is used for mobile applications with limited memory.

The LRU cache may have a maximal capacity so that the oldest, never used entries are discarded. Furthermore, expired advertisements will be removed automatically.

The discovery protocol is designed to always differ between peer, group and other types of advertisements. We therefore have three LRU cache objects defined in the discovery service, one for each such type.

Advertisements can be accessed by their unique ID or by a key/value search which make use of the advertisements' `match` method. It is also possible to get a number of random advertisements (or all of them).

Advertisements are identified by its unique ID defined in the advertisement. When creating a new advertisement type, it is important to define a good scheme for the unique ID. While it is not always possible to create a unique ID because the advertisement information is too common, we have to be aware that an advertisement cannot be stored in the cache without having a unique ID.

The cache itself provides also the possibility to extract an SRDI for all or for advertisements as of a certain point in time. For this, it uses the advertisements' `index_elements` method.

No capacity limits There is a general drawback regarding the limited capacity in the cache in combination with the minimal implementation of the endpoint router.

Because the endpoint router is not able at the moment to automatically query for route advertisements[4], the advertisements cache gets useless when it removes seldom used peer and route advertisements from time to time. After all, the peer application should not always have to check whether the recipient's route advertisement is available or not. It should be possible to query once for a peer advertisement[5] and then to just send messages without worrying about the route.

Therefore, we have abolished the capacity limit for the moment. Of course, the challenge with advertisement expiration remains. The peer application has to guarantee that advertisement updates are published regularly to the peer group.

4.8 Shared creators

There are some important creators (also called factories) in *VamPeer*. All creators can be accessed using the class `P2P_CREATORS_SHARED`:

[4]The router needs the recipient's route advertisement in order to resolve the peer ID to a transport address.

[5]The peer advertisement contains also the peer's route advertisement.

ID creator The ID creator represented by `P2P_ID_CREATOR` provides globally unique instances for the null, NPG and WPG ID and provides the possibility to parse an ID string and to create the appropriate ID object out of it.

It knows and can handle the `jxta` and `uuid` format types. Custom creators may be registered. Such an agent gets a generic ID and has to return a specific ID object or `Void` if the ID is not recognized. The custom creators can override the *VamPeer*'s types.

XML document creator Specific XML documents are also derived with a shared creator: `P2P_XML_DOCUMENT_CREATOR`. There are mainly two methods to create XML documents. Both expect the XML root element.

The first method `document_from_element` returns a `P2P_XML_CACHE` object or `Void` whenever the creation was not successful. If the document type cannot be detected, it returns a valid `P2P_UNKNOWN_XML_DOCUMENT`.

It is used by the second method `advertisement_from_element` which simply tries to cast the result to an advertisement (`P2P_ADVERTISEMENT`).

The document creator is also useful for user services as custom document types can be registered too, similar to the ID creator.

Wire message creator The last creator is used for the endpoint message's wire representation. Currently, there is only one mime type used (`application/x-jxta-msg`). Thus, the wire representation is always in binary format, handled with `P2P_WIRE_MESSAGE_BINARY`.

5 Usage

In this chapter, we are looking at the API and present how the *VamPeer* library can and should be used. While we first list all the possibilities, we describe also some examples which are all supplied with the library. The next chapter then focuses on a bigger example, a simple Origo cluster.

5.1 Platform starting

For starting and stopping the platform, we always deal with a `P2P_PLATFORM` instance. To get the instance, we call `make` passing the configuration directory path and a logger (`L4E_LOGGER`). The platform automatically reads the configuration file `PlatformConfiguration.xml`, if available, and sets the `is_configured` variable appropriately. If it is false, we have to provide a new `P2P_CONFIGURATION` as shown in listing 5.1.

```
configure_platform is
    -- Configure VamPeer platform instance
  require
    Logger_valid: logger /= Void
  local
    conf: P2P_CONFIGURATION
  do
    create platform.make (".vampeer", logger)
    if not platform.is_configured then
      conf := platform.default_configuration -- New NPG Peer
      conf.set_name ("Peer name")
      conf.set_description ("Peer node description")
      platform.configure (conf)
    end
  ensure
    Platform_set: platform /= Void
  end
```

Listing 5.1: Configuring the platform instance

The default configuration returns the settings for a new NPG peer: A new peer ID is created, the TCP settings are set to automatic interface/port detection and the rendezvous is set to client mode with the standard NPG rendezvous servers. Before the configuration is fed into the platform, we set an appropriate peer name and a description.

The next step is to load the NPG. For the public NPG, this is easily done with the `standard_net_peergroup` method which takes care of module implementation advertisement creation and group loading.

```
start_platform is
    -- Load and start net peer group
  require
    Platform_configured: platform /= Void and platform.is_configured
  do
    npg := platform.standard_net_peergroup  -- Load NPG
    if platform.module_status /= platform.init_failed then
      platform.start  -- Start Platform/NPG
    end
    if platform.module_status /= platform.start_ok then
      npg := Void
    end
  ensure
    Npg_set: npg /= Void implies npg.module_status = npg.start_ok
  end
```

Listing 5.2: Loading/starting the platform with the public NPG

Modules can make use of start parameters which are provided with the platform's `start_with_arguments` command (expecting an `ARRAY [STRING]`). All started modules will receive the same arguments. However, the JXTA services do not actually use them.

When the platform is successfully started, we may access the services via the NPG peer group instance. Note that at this point of time, the rendezvous client may not yet be connected to a server and propagated messages will be lost. See subsection 5.2.3 to see solutions to this problem.

We should not forget to maintain an event loop when we are done with initializing, or the application will quit instantly. The platform does not provide such a feature.

To stop the platform again, we just call `platform.stop`. If you just like to pause the P2P activity, you will like the `suspend` command (see section 3.2).

5.1.1 Private peer groups

It is somewhat more complicated to load a private peer group since we have to provide and create more specific settings. The entire process looks very similar but we have to create another configuration and load the private NPG differently.

Creating new IDs The very first step is to create all the IDs used for the new peer group. Please read the introduction in section 3.3 at page 25 to get an overview about the definition of a peer group.

Listing 5.3 shows how to generate the three IDs used for a new private NPG: a MCID and a MSID for the peer group module and finally the peer group ID. A private NPG is always a child of the WPG, so we use the WPG UUID as parent ID.

```
create_peergroup_ids is
    -- Create all IDs used for a new peer group
  local
    mcid: P2P_MODULE_CLASS_ID
    msid: P2P_MODULE_SPECIFICATION_ID
    gid: P2P_PEERGROUP_ID
    wpgid: P2P_WORLDGROUP_ID
  do
    create mcid.make_new
    create msid.make_new_with_class (mcid.uuid)
    create wpgid.make
    create gid.make_new_with_parent (wpgid.uuid)
  end
```

Listing 5.3: Creation of IDs for a new peer group

You will not use that code in your peer application; the creation of the new IDs is a one-time process. You will hard code the new IDs in your application afterwards. Instead of using this example above, you would better use the *idcreator* example which generates all the different kinds of IDs.

Creating a configuration To create an appropriate configuration, we have to set the peer ID, name and description in a `P2P_CONFIGURATION` object as shown in listing 5.4. Additionally, we have to add service configurations for the tcp and the rendezvous module. For the TCP configuration, you may safely choose the default one with `default_tcp_configuration`. The rendezvous configuration needs to be adapted to the groups rendezvous servers, see subsection 5.2.3 for more details.

```
new_configuration: P2P_CONFIGURATION is
    -- New configuration for a private peer group
  require
    Group_id_valid: gid /= Void and gid.is_valid
  local
    pid: P2P_PEER_ID
    rdvconf: P2P_RENDEZVOUS_CONFIGURATION
  do
    create pid.make_new_with_group (gid.uuid) -- New peer ID
    create rdvconf.make -- Rendezvous client configuration
    rdvconf.add_seed_uri ("http://stablehost.org/rdvs.cgi")
    create Result.make_with_id (pid)
    Result.add_service_parameter (transport_tcp_mcid,
        default_tcp_configuration)
    Result.add_service_parameter (rendezvous_mcid, rdvconf)
  ensure
    Result_set: Result /= Void and Result.is_valid
  end
```

Listing 5.4: Creating a platform configuration for a private peer group

The constants for the TCP and rendezvous MCIDs can be found in P2P_CONSTANTS.

Creating a module implementation advertisement A module implementation advertisement for the group should be created when it is not available in the cache yet. Listing 5.5 shows how we first can get a standard advertisement containing already all advertisements for the JXTA services. We then add a user service and store the entire document to disk using the cache manager.

```
set_peergroup_implementation_advertisement is
    -- Make sure that peer group implementation advertisement exists
  require
    Platform_valid: platform /= Void
  local
    params_doc: P2P_XML_DOCUMENT
    pg_mia, smia: P2P_MODULE_IMPLEMENTATION_ADVERTISEMENT
  do
    if not platform.cache_manager.has_module_implementation_advertisement (
        pg_msid) then
      pg_mia := platform.peergroup_implementation_advertisement (pg_msid, "
          PG_CLASS", "group description")
      params_doc := Result.parameter.document
      -- Add user service impl adv
      smia := platform.default_implementation_advertisement (service_msid,
          "SERVICE_CLASS", "service description")
      params_doc.create_root_child_element ("Svc", namespace_empty)
      params_doc.add_child_element (params_doc.last_element, smia.document.
          document.root_element)
      -- Store group impl adv
      platform.cache_manager.store_module_implementation_advertisement (
          pg_mia)
    end
  end
```

Listing 5.5: Creating a peer group module implementation advertisement

Loading a private NPG To load a private peer group, the platform provides the method load_net_peergroup which expects the group ID, the specification ID and an agent for instantiating the group.

We do not like to load the peer group module directly because we want to have a unified access through the platform object. This also allows the platform to be up to date with its module status.

```
load_peergroup is
    -- Load a private net peer group
  do
    npg ?= platform.load_net_peergroup (gid, pg_msid, agent
        peergroup_loader)
  if npg /= Void and npg.module_status /= npg.init_failed then
    npg.group_advertisement.set_name ("Group name")
    npg.group_advertisement.set_description ("group description")
    platform.cache_manager.store_peergroup_advertisement (npg.
        group_advertisement)
  end
end

peergroup_loader (a_pg: P2P_PEERGROUP; an_id: P2P_ID; a_mia:
    P2P_MODULE_IMPLEMENTATION_ADVERTISEMENT): P2P_MODULE is
  -- Private NPG loader
do
  if a_pg = platform and an_id.is_equal (gid) and pg_msid.is_equal (a_mia
      .specification_id) then
    create {PG_CLASS} Result.init (a_pg, an_id, a_mia)
  end
end
```

Listing 5.6: Loading a private NPG

5.2 Services

After starting the platform, we will only work with the peer group instance and its services. All modules are registered in the peer group, so we may access them through our NPG. While the JXTA services have an easy access method (for example `npg.endpoint_service`), we have to access user services through their module name: `npg.lookup_module("servicename")`.

5.2.1 Endpoint service

The endpoint service is very central because all other services rely on it, directly or indirectly. This is true on the one hand for all services and on the other hand also for the transport modules.

Receiving endpoint messages We shall first look how services may use the endpoint service. When they are ready to receive endpoint messages, they register a service name, an optional service parameter and a handler using `extend_service`. The service name is usually the assigned module ID of the registering service, therewith we get a unique ID.

The actual, internal handler name is the combination of the service name and the parameter. When we specify also a parameter, we get only those messages that match exactly the service name *and* the parameter. If no such handler exists, the handler matching only the service name will be called. Note that we can only have

```
process_message (a_msg: P2P_MESSAGE; a_source, a_destination:
   P2P_ENDPOINT_ADDRESS) is
    -- Process incoming endpoint message
  require
    Message_valid: a_msg /= Void
    Source_valid: a_source /= Void
    Destination_valid: a_destination /= Void and a_destination.service_name
       .is_equal (sname)
  local
    msgel: P2P_MESSAGE_ELEMENT
  do
    msgel := a_msg.element_by_namespace_and_name (a_msg.namespace_user, "
       dummy")
    if msgel /= Void then
      print (msgel.content)
    end
  end
```

Listing 5.7: Example endpoint message handler

one handler per service; when a handler is registered, a possibly old handler for the same service name/parameter is silently replaced.

Message handlers get an endpoint message together with the extracted source and destination endpoint address. Handlers should not do long processing jobs and should return as soon as possible[1].

See listing 5.7 for an example message handler. It prints out the content for the message element with name “dummy” from the user namespace.

Reaching remote peers For actively reaching other peers, there are three possibilities:

1. `send_message` and `send_message_mangled`,
2. `propagate` and `propagate_mangled` and
3. `ping`.

To send a message, we simply call one of the send message methods passing an endpoint message together with an endpoint address. The endpoint address should use the protocol `jxta` with a peer ID. Listing 5.8 shows a simple example for this.

To learn if you need the “mangled” version or not, you may revert to section 3.5 on page 29. However, using `send_message_mangled` is usually the safer way, but requires us to pass the group ID of the calling service.

[1]See section 4.6.1 about the thread handling in our TCP transport.

```
send_endpoint_message (a_dest: P2P_PEER_ID) is
    -- Send endpoint message to ‘a_dest’
  require
    Dest_valid: a_dest /= Void and a_dest.is_valid
  local
    ea: P2P_ENDPOINT_ADDRESS
    msgel: P2P_MESSAGE_ELEMENT
    msg: P2P_MESSAGE
  do
    create ea.make_with_id (a_dest, "pingservice", Void)
    create msg.make
    create msgel.make_string (msg.namespace_user, "Data", Void, "Ping")
    msg.extend (msgel)
    peer_group.endpoint_service.send_message (ea, msg)
  end
```

Listing 5.8: Creating and sending an endpoint message

The propagate methods pass the message to all transports which should make use of transport specific propagation techniques. Currently, only the TCP transport can handle this request by sending a UDP multicast packet[2]. However, our TCP module only implements outgoing multicast and cannot read any incoming multicast messages. Thus, multicast is virtually useless at the moment.

Using the endpoint's propagation method will at best reach peers in the local network, never the entire peer group. To propagate a message, we do not pass a full destination address because it is protocol and destination unspecific; we just pass the destination's service name and parameter.

While the methods for sending or propagating messages are normally asynchronous, the ping command is a time-consuming call as it waits for a remote answer. We do not send messages with `ping`, we just like to check whether the given endpoint address is valid and available to us or not. The TCP transport for example tries to open a connection and to do a handshake with the remote peer.

Transport handling Transport modules also register with the endpoint service but using the method `extend_message_transport`. Transports may either be responsible for incoming (`P2P_MESSAGE_RECEIVER_TRANSPORT`) or outgoing messages (`P2P_MESSAGE_SENDER_TRANSPORT`) or both; the endpoint service is able to deal with these types.

Transports may induce messages by calling the `demux` command which analyzes the message and passes it to the appropriate service. The endpoint service will simply ignore messages for which no service handler exists.

[2]The message size is therefore limited to 16KB.

Message filtering The endpoint service has the feature to filter messages out. It is possible to extend filters for incoming or outgoing messages. Filters may not only decide whether a message is discarded or not but may also change the messages. Listing 5.9 shows how such a filter handler could look like.

```
message_filter (a_msg: P2P_MESSAGE; a_src, a_dest: P2P_ENDPOINT_ADDRESS):
    P2P_MESSAGE is
    -- Discard incoming messages, if 'ignore_all' is set
  require
    Message_valid: a_msg /= Void
    Source_valid: a_src /= Void
    Destination_valid: a_dest /= Void
  do
    if ignore_all then
      logger.info ("Discarding message from: " + a_src.out)
    else
      Result := a_msg -- feed message back to other filters and services
    end
  ensure
    Result_set: ignore_all = (Result = Void)
  end
```

Listing 5.9: Example endpoint message filter

5.2.2 TCP Transport module

For the TCP transport module, we will not show its interface because a user never gets into direct contact with the module's features. But we like to show how the user may configure the transport.

The TCP transport looks for its service parameter in the platform configuration. The configuration must be existent or the module will not start. There are currently three values to set: the `Port`, the `InterfaceAddress` and the `MulticastOff` flag.

```
<Parm type="jxta:TCPTransportConfiguration">
  <MulticastOff></MulticastOff>
  <Port>9701</Port>
  <InterfaceAddress>129.132.105.170</InterfaceAddress>
</Parm>
```

Listing 5.10: Full TCP transport configuration

When the *MulticastOff* flag is set, the TCP module will disable propagating via multicast. When the flag is not set, only sending of multicast messages is supported. The module currently does not support listening for multicast messages. We should therefore disable multicast.

The *port* specifies the server port. If it is not set, the module will automatically choose a port in the range 1024–65'535.

The *interface address* also concerns the TCP server. When the address is set, the module will only accept messages for the given address. We may use this in combination with VPN (virtual private network) to lock out messages from unauthorized sources.

When the interface address is not specified, the module tries to detect the interface and only listens on this interface. It is important for the platform to know its own IP, namely to create the peer advertisement which includes the route information.

The platform can not always detect its IP correctly. It has to use a connection to find out the local IP. This is done when the rendezvous seed URL is resolved when the platform is started (see the next sub section). When the platform does not need to get the rendezvous seed list, it will set the local IP to `127.0.0.1`.

5.2.3 Rendezvous service

The essential rendezvous service interface is rather small as it only has to provide methods to propagate messages[3]. The main part of the entire server connection handling is done internally and is not really of interest for a user.

Message propagation When propagating a message, one has to provide the endpoint message, the destination's service name/parameter and a TTL.

The TTL is an integer value meaning the maximal number of hops the message can be forwarded. Usually, we just set the maximum `Ttl_max` (50).

There are several propagation methods: `propagate_in_group` sends the message to all connected rendezvous servers whereas `propagate_to_neighbours` uses the endpoint's propagation method (propagating to the local network). `propagate` is usually the preferred method as it calls both methods above. We provide also the possibility to propagate messages to a given list of peers with `propagate_to_peers`.

The rendezvous service is only responsible for sending messages, the recipient's endpoint service will pass the message directly to the specified service, the rendezvous service is not touched at the recipient side.

When we repropagate a message (meaning to propagate a received propagated message), the rendezvous service is able to detect this and automatically reuses the meta data stored in a special rendezvous message element.

Rendezvous events As many services rely on message propagation, they would like to make sure that the current peer is connected to a rendezvous so that message propagation is guaranteed. However, just at the time when the platform has started, the rendezvous connection is not available yet. So, it is rather useless at this time to use the service.

We therefore need a way to know when the connection will be of use to the services. That is what *rendezvous events* are designed for. Interested parties may register for such events. Current supported rendezvous types are only the connection and

[3]See section 2.4.3 for a detailed rendezvous service description.

disconnection events to a rendezvous server as we currently just implement an edge peer.

To receive these events, we register an agent with `extend_rendezvous_event_handler`. The agent should expect a `P2P_RENDEZVOUS_EVENT` which provides the event type and the involved peer ID (e.g. the rendezvous server). An agent should not do time consuming processes as it would stall incoming messages from the rendezvous. Listing 5.11 shows how to publish the peer advertisement as soon as we are connected to the group.

```
process_rendezvous_event (an_event: P2P_RENDEZVOUS_EVENT) is
    -- Publish our peer advertisement to group when connected to rdv
  require
    Event_valid: an_event /= Void
  do
    if an_event.type = {P2P_RENDEZVOUS_EVENT}.type_connected_to_rendezvous
       then
      peer_group.discovery_service.publish_advertisement_remotely (
          peer_group.peer_advertisement, Void)
    end
  end
```

Listing 5.11: Example rendezvous event handler

Rendezvous Seeds We somehow have to specify the rendezvous server address (or multiple addresses) so that a new, isolated peer can contact the group. For a peer application, one would elect some peers as permanent rendezvous servers and make their addresses available.

As described in section 2.4.3, there are several ways to do this. The preferred solution is to maintain a file, accessible through HTTP, with a rendezvous server list. The peer's configuration then would be hard coded to this URL. The NPG rendezvous list can be found at the following address, listing 5.12 shows its content:
`http://rdv.jxtahosts.net/cgi-bin/rendezvous.cgi?2`

```
http://209.128.126.120:9700
http://209.128.126.120:9710
tcp://192.18.37.36:9701
tcp://192.18.37.37:9701
tcp://192.18.37.38:9701
tcp://209.128.126.120:9701
tcp://209.128.126.120:9711
```

Listing 5.12: Public NPG rendezvous seeds

Edge peers then randomly try one of the seed addresses and continue trying until they get a connection lease.

The rendezvous configuration contained in the platform configuration file is simple and looks like the one in listing 5.13. *VamPeer* updates the configuration with known servers, once it has resolved a seed URL.

```
<Parm type="jxta:RdvConfig" config="client">
  <seeds>
    <addr seeding="true">http://origo.ethz.ch/rdv.cgi</addr>
    <addr>tcp://129.132.105.170:9700</addr>
  </seeds>
</Parm>
```

Listing 5.13: Example rendezvous configuration

5.2.4 Resolver service

Resolver handlers Clients using the resolver service choose a unique handler name and register a handler agent for processing query and one for response messages. Queries and responses are thus tightly coupled because when you send a query, you are interested in replies and a typical peer will perform queries and replies. The handler registration works with the `extend_handler` command. The handler name is usually the client service module ID.

A resolver query handler gets a `P2P_RESOLVER_QUERY` object containing the query string with some meta information. The agent should set the `repropagate` flag in the query object to specify whether the resolver should repropagate the message or not. Repropagation is only done by the resolver if the peer is configured as a rendezvous. The response handlers have the equivalent signature, they expect a `P2P_RESOLVER_RESPONSE` object.

Besides queries and responses, the resolver handles also SRDI messages. It is basically the same as with queries, so we overlook the exact details here.

Querying and responding To send a query, we build a `P2P_RESOLVER_QUERY` as shown in listing 5.14[4]. Beside the query string, it needs the source peer ID, the handler name and an integer ID. The ID's purpose is to help identify responses. The resolver automatically adds the source peer route advertisement so that every recipient is able to respond directly.

```
send_resolver_query (a_dest: P2P_PEER_ID) is
    -- Send a resolver query
  require
    Dest_valid: a_dest /= Void and a_dest.is_valid
  local
    query: P2P_RESOLVER_QUERY
  do
    create query.make (peer_group.peer_id, handler_name, 1, "Got a beer?")
    peer_group.resolver_service.send_query (a_dest, query)
  end
```

Listing 5.14: Sending a resolver query

[4]The final query would then look like in listing 2.2 on page 19.

The example sends the query to a specific peer but it is also possible to send messages to the entire group. We just have to use the propagate_query method which only expects the query.

Responding is equivalent. It is somewhat easier to respond to a received query because P2P_RESOLVER_RESPONSE provides a constructor to create a response out of a query: make_from_query.

Note that a response does not need to be preceded by a query. This means that a response can also be propagated in order to publish information to everyone.

5.2.5 Discovery service

As we already know is the discovery service used to deal with advertisements. It provides methods for querying and publishing, locally and remotely. Most functions differ between peer, group and other (general) types of advertisements because also the discovery protocol does.

Querying A query always consists of the advertisement type and possibly a key/value pair. The value may be unspecified in local queries meaning to find all advertisements that have an element named like the key. Wildchars are also allowed in values as already described in section 2.4.5.

For local queries, we call one of the local_*_advertisements methods (there is one for each type) passing the key and a value. The return value is a list of all matching advertisements.

It is also possible to get an advertisement from the store using its unique ID. This is done with a local_*_advertisement command. While you pass an ID for the peer or group advertisements, you have to specify the exact unique ID as STRING when looking for an other advertisement type.

Local queries can always be performed except when the discovery module is stopped or has failed during the start.

Remote queries are done with the query_remote_advertisements method. For this, we need to build a P2P_DISCOVERY_QUERY object and possibly specify a single recipient peer and a response handler. When the recipient peer is not specified, the query is propagated to the group.

A query object may contain an advertisement type, a key/value pair, a threshold and a source peer advertisement. We have to pay attention to the different semantics for some attribute combinations.

Normally, we specify all the attributes which means to search for all matching advertisements of the given type. The threshold defines how many results we like to receive in maximum. However, we could receive a lot more since we may receive answers from various peers. A peer query example is provided in listing 5.15 on the next page.

Specifying only the type Peer and the threshold 0, all recipient peers should send a response with their own peer advertisement.

```
discover_buddy is
    -- Send discovery query for peers named "Buddy"
  require
    Network_connected: peer_group.rendezvous_service.is_connected
  local
    query: P2P_DISCOVERY_QUERY
  do
    create query.make (peer_group.discovery_service.type_peer)
    query.set_threshold (10)
    query.set_restriction ("Name", "Buddy")
    peer_group.discovery_service.query_remote_advertisements (query, Void,
        agent response_handler)
  end

response_handler (a_response: P2P_DISCOVERY_RESPONSE) is
    -- Prints buddy IDs of all results
  require
    Response_valid: a_response /= Void and a_response.is_valid
  local
    advs: DS_LIST_CURSOR [P2P_PEER_ADVERTISEMENT]
  do
    from
      advs := a_response.all_peer_advertisements.new_cursor
      advs.start
    until
      advs.after
    loop
      print (advs.item.peer_id.out + "%N")
      advs.forth
    end
  end
```

Listing 5.15: Sending a remote discovery query

When the key/value pair is not specified, recipient peers should return a random advertisement set matching the given type. The set's count should not exceed the threshold value.

Response handlers In the discovery service, there are two possibilities to add a response handler. If we are only interested in our query, we specify a *query ID* handler when querying. This handler will only be called when a response for our query is received. Since multiple responses can be sent, the handler stays registered until we call `remove_queryid_listener`.

The other response handler type serves for general responses which can be registered through `extend_response_listener`. Whenever a response is received, the discovery calls all response agents (actually, after a possible, specific query ID handler).

When no response handler is called, the discovery will itself locally publish all received results. So, whenever we register a handler, we have to handle the results and publish them, if needed.

It is only possible to register handlers for responses. Queries are always handled by the discovery service itself. We can not tune the discovery to respond differently to certain queries.

Publishing Publishing an advertisement locally, using `publish_advertisement_locally`, means to save it in the local advertisements store under its unique ID. An older advertisement with the same ID is replaced.

On remote publishing, we may either choose to build the discovery response ourselves or to just pass a single advertisement to `publish_advertisement_remotely`. We also may specify a recipient peer, when we do not want the response to be published to the group.

A `P2P_DISCOVERY_RESPONSE` contains besides a list of results, the type and the key/value pair (if it is really a response). A respondent may also provide its peer advertisement.

An advertisement is always published together with its remote expiration time. When the lifetime is not set, the default expiration time for remote peers is by default two hours.

Note that remotely published advertisements cannot be revoked. They may be passed among group peers until they expire. Though, it is possible to delete an advertisement locally using one of the flush methods which expect the advertisement's unique ID.

See listing 5.16 to see a publishing example and how to set the lifetime.

```
publish_advertisement (an_adv: P2P_ADVERTISEMENT) is
    -- Publish an advertisement setting its lifetime to a day from now
  require
    Advertisement_valid: an_adv /= Void and an_adv.is_valid
  do
    an_adv.set_lifetime_relative (86400000)
    peer_group.discovery_service.publish_advertisement_remotely (an_adv,
        Void)
  end
```

Listing 5.16: Publishing an advertisement remotely

5.3 Writing a P2P application

While the *VamPeer* library offers very basic P2P features, the overlying applications will have to specialize them and to design their own application messaging protocol above JXTA.

For its design, we recommend to rely on the JXTA services, especially on the discovery. The use of advertisements is a core idea in JXTA and it would be great to create advertisements for user related entities too.

Unfortunately, there is a problem associated with this endeavour since the rendezvous server has to know all advertisement types it deals with. This means that we currently either have to bypass the rendezvous or to implement user defined types in the JSE rendezvous peer too.

5.3.1 Writing a user service

Applications will usually create at least one service which closes the gap between the application's logic and JXTA.

To create a module, we just inherit from the deferred class `P2P_MODULE` and effect the methods `start`, `suspend` and `stop`[5]. We usually also redefine the `init` method to create needed data structures.

We also have to implement the feature `check_dependencies` which has the purpose to identify the required module dependencies used for loading. It is therefore called in the precondition of `init`. We are allowed to use also other services but we then have to check them for existence first.

While it is simple to build a user service, the integration into the peer group involves several steps.

As the user service is a module, we have to build a MCID, a MSID and a module implementation advertisement. Listing 5.5 already pointed out how to do this.

Because we change the set of user services in the group, we will certainly have to build a private peer group as described in section 5.1.1. We now also have to specialize the peer group implementation by creating a new peer group class which should inherit from `P2P_GENERIC_PEERGROUP`. `define_modules` and `load_extern_module` should be redefined as shown in listing 5.17.

```
define_modules is
    -- Define Group services in 'modules_list'
  do
    -- add standard modules from parent peer group to 'modules_list'
    Precursor
    -- add user service to the end of 'modules_list'
    modules_list.put_last (["user_service", service_mcid, service_msid,
        parent_is_owner_if_available])
  end

load_extern_module (an_id: P2P_ID; a_mia:
    P2P_MODULE_IMPLEMENTATION_ADVERTISEMENT; a_name: STRING): P2P_MODULE is
    -- Load user module
  do
    if service_msid.is_equal (a_mia.specification_id) then
```

[5]Please read section 3.2 for further details.

```
      create {SERVICE_CLASS} Result.init (Current, an_id, a_mia)
    else
      Result := Precursor (an_id, a_mia, a_name)
    end
  end
```

Listing 5.17: Redefining peer group modules

The `parent_is_owner_if_available` flag means that the module loader will only create a new module loader if the parent peer group did not define this module. If each peer group would need its own instance, we would specify `current_is_owner`.

Using the new private peer group implementation, our user service will be loaded and managed by the *VamPeer*'s platform.

5.4 Examples

The current *VamPeer* release also contains some examples so that new users are able to get into the code and can test the library. We will describe them here and show how to run them. There is also a bigger example which we will separately enlighten in the next chapter.

5.4.1 Endpoint message sender/handler

The first example shows how to use the *endpoint service*. It was originally created when the other services were not available yet. Therefore, it does not make use of the discovery to find other peers.

It consists of two parts, namely two peer roles: the `endpoint_message_sender` and the `endpoint_message_handler`, which both run in the public NPG. The sender will send a simple endpoint message to the handler peer which sends a reply message back. Both peers log the events to the standard output, so we can see what is currently happening. While the handler only quits when we shut it down (with `Ctrl-C`), the sender terminates as soon as it receives the reply.

The handler's platform is configured to listen on port `9710` while the sender chooses its port automatically. To start the sender, we have to specify the handler peer's IP and port. This is of course not the way we would like to deal with other peers in our application. We just like to limit our example to the endpoint service's possibilities.

To run the example locally, we first start the endpoint message handler and afterwards the sender application with the arguments: `localhost 9710`. The logging level is set to `INFO`, so we can see when the platform has been started and the message has been sent and received.

Looking at the code, we see how the sender creates and sends an endpoint message and how it registers the service listener to receive the reply. The handler looks similar but also uses the endpoint's filter method to display all incoming messages.

The example is very simple but shows how the minimal configuration for a *VamPeer* platform may look like.

5.4.2 Rendezvous propagation

The next example demonstrates the rendezvous propagation mechanism. It resides in the `rendezvous_propagate` example directory. The application connects to a NPG rendezvous and propagates a message every five seconds.

When running multiple instances, we could see the propagated messages from the others. But it does not work; messages get not propagated to the entire group. The reason for this is that we would have to adapt the rendezvous server to repropagate messages for our specific service[6].

Thus, this example shows how we can not use the rendezvous service for propagating any message and it shows how important it is that the rendezvous server is part of the P2P system and not just a standard JXTA infrastructure peer.

5.4.3 Discovery

The `discovery` example shows how we can find other peers in the public NPG. We just run multiple instances and they should be able to discover each other. The application only expects a configuration directory path as argument.

The details of the discovery procedure are as follows: As soon as we are connected to a rendezvous, we publish our peer advertisement in the group. We then send one peer query request and wait one minute for responses. Incoming peer advertisements are printed out immediately.

When no other peers are known to the rendezvous, we should at least receive the rendezvous peer and our own advertisement. Note that we may get responses from multiple peers, not only from the rendezvous because the rendezvous propagates the query request also to the others.

5.4.4 JXTA JSE rendezvous server

The last two examples make use of the NPG rendezvous servers. But we mentioned already that the public infrastructure is not accessible at the moment. Hence, we will have to run our own NPG rendezvous server. The `RdvServer` example shows how we can set it up using the JXTA's reference implementation JSE in version 2.4.1.

The Java application provides a rendezvous and a relay server and is configured to listen on port `9700` on all interfaces. When we start the script `startNPGPeer.sh`, the rendezvous will be started in the NPG. Because we use also a rendezvous peer for the next example, namely Origo, there exists another script `startOrigoPeer.sh` which starts the rendezvous in the private Origo peer group.

To point the discovery example to our new rendezvous server, we have to change its platform configuration. We normally start the discovery application first to generate the configuration directory. Then, we are able to change the configuration XML file by replacing the old seeds and the NPG seed URL with the new destination address.

[6]See section 3.6 for more details to this problem.

6 Origo with VamPeer

As a case study, we tried to build a P2P application with *VamPeer*. We supply it as an additional example.

Because our task was to prepare P2P support for Origo, it was obvious to do a test application for Origo. We did not implement the real Origo messaging layer because the full requirements were not yet available at this time[1]. We did therefore a simple Origo example which supports some core ideas of Origo, especially regarding the P2P technology.

We will now give an Origo overview so that we understand what our example deals with.

6.1 Origo overview

Origo is a PhD project by Till Bay, the supervisor of this master thesis. Origo is a revolutionary software development, management and distribution platform. While the main idea exists, the implementation just has started in the last few months; *VamPeer* is also a contribution to it. We hope that in a few months, we have a first Origo release supporting the main features.

Many similar platforms exist (SourceForge, BerliOS, Google Project Hosting, . . .). The added value by some platform is to provide techniques such that the components may cooperate. For example when a patch is committed solving a certain bug, the bug tracker should be able to detect this event and close the corresponding issue (this could be done by parsing the commit log entry).

While some platforms already support very good services, Origo still is the only one which will support the following essential features: modularity and extendability.

The traditional platforms like SourceForge are rather monolithic since their various services are coupled together in a tight, static manner. Thus, it is not a simple task to add a new version control system or a bug tracker service without having to change the system essentially.

That is the reason for Origo to support modularity. Very simplified, this is done by defining a messaging layer which all components will have to use. They share messages about what features and event types they provide and may thus cooperate together. This is why each component has some kind of a wrapper which integrates the component into the Origo system.

[1]Patrick Ruckstuhl currently writes a master thesis about the *Origo Core* where he will among others define and implement the entire messaging layer using *VamPeer*.

All the resulting Origo services are managed by a central *core* service. It provides also an XML-RPC API to the public so that central features as creating a new release could be integrated into the developer's environment.

Origo's main services will be revision control, bug tracker, wiki, single sign-on and lookup services to find appropriate software components.

Origo wants to scale under heavy load; Origo is therefore designed to run as a distributed system. When new services are added dynamically to the system, they must be able to discover and contact the core service. Thus, it is obvious to use P2P technology for messaging.

As Origo is fully developed in Eiffel, we now can see the impulse for our master thesis which brings a P2P system, namely JXTA, to Eiffel so that it can be used in Origo.

6.2 Design

We now look at our Origo example and what features we support. The official Origo release will be based on another design. We just show how we can handle the network challenges with *VamPeer*.

6.2.1 Task

In our example, we focus on the task how services can advertise their existence to the core service and how the core service may start the Origo infrastructure. We only have two peers: the core peer and a service peer which provides a lookup service[2].

Figure 6.1 shows the example's *VamPeer* related classes and their interfaces.

An Origo service in our example is advertised through `O_SERVICE_ADVERTISEMENT`. A service has several roles which are each represented through a `O_SERVICE_ROLE_ADVERTISEMENT`.

Services may have several roles because a service itself could be distributed too and may consist of several components. An `O_SERVICE` object is the local representation of the entire service and manages all the roles that are registered at the current peer.

The lookup service has two roles: a main node responding to queries and a proxy node which resides on the core node to forward queries to the main node. The initial idea for the proxy was to hide the real lookup peer from the public network so that each request must go through the core peer which could also maintain a special security layer.

The core's task is to know all services and also the status of all its roles. When a new service joins the Origo peer group, the core peer looks for all service roles and when they are all available, a request is sent to all involved peers to start and integrate the entire service into the Origo system.

[2]The lookup service implementation is managed by Andrea Grössbauer who writes a semester thesis about hierarchical lookup queries.

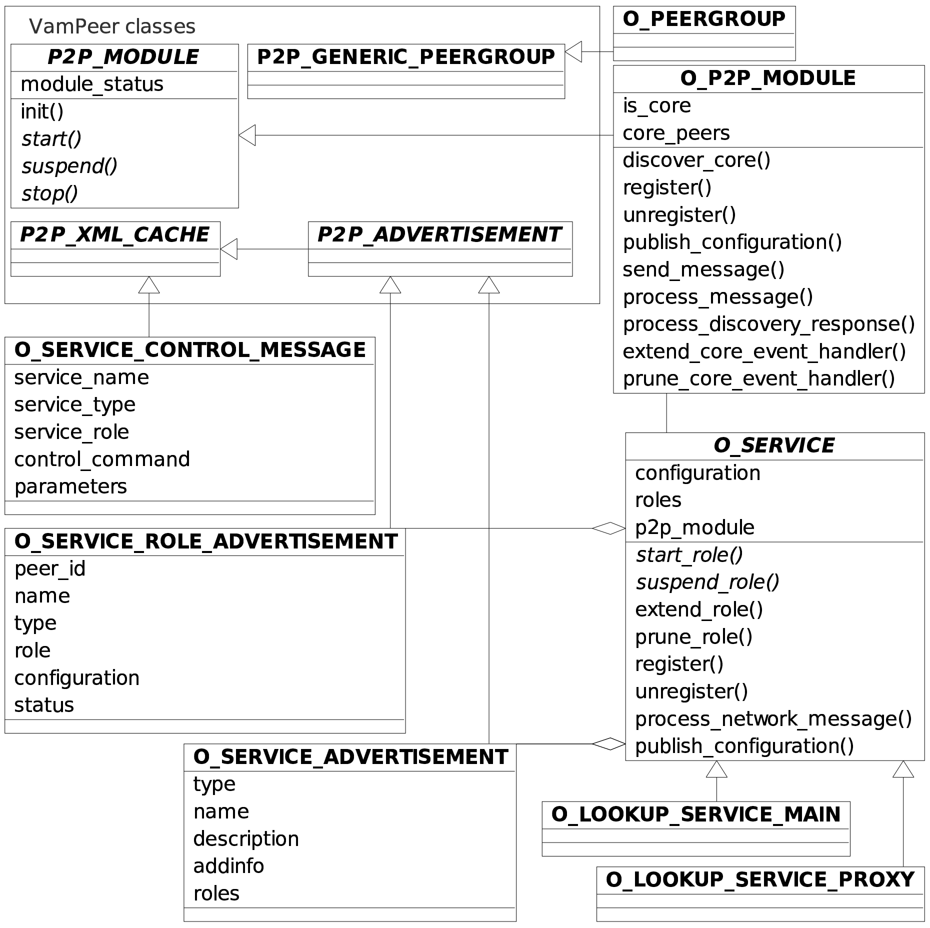

Figure 6.1: Origo example P2P class structure

6.2.2 P2P infrastructure

The Origo example runs in a private peer group, the Origo group. The `RdvServer` example described in section 5.4.4 also supports this group.

The group uses all *VamPeer* services and additionally introduces a new service module implemented by the class `O_P2P_MODULE`. This module is used by the Origo services[3] to communicate with other peers or vice versa.

The communication between peers is on the one hand done with discovery messages to share advertisements and on the other hand with XML control messages (`O_SERVICE_CONTROL_MESSAGE`) based on endpoint messages.

We can use Origo specific advertisements because we do not share them with the rendezvous server, we pass them directly to the appropriate peers.

[3]Note the difference between a JXTA/*VamPeer* service and a Origo service.

6.2.3 Startup procedure

The core peer must be started first and publishes its peer advertisement to the peer group. Other peer nodes just propagate a core discovery query.

When a peer node starts, all Origo services register at the network module, so it knows what services and service roles are available locally.

As soon as it receives the core's peer advertisement, it publishes all the registered service and role advertisements to the core.

The core checks on each incoming role advertisement whether all roles for the given service are available now. When they are complete, the core sends a start control message to all peers which maintain an involved service role.

Once a peer receives this message, it advises the corresponding service object to start. The lookup proxy node will register its XML-RPC proxy methods on the local XML-RPC server whereas the lookup main node will start up the lookup component.

6.2.4 Role configurations

The roles may have to exchange their configuration parameters. Thus, each service role advertisement can have a configuration XML document attached, for example a descendant of `O_SERVICE_CONFIGURATION`.

The lookup service for example uses this configuration to tell the proxy on which IP address and port the lookup main component is listening to answer XML-RPC queries.

6.3 Summary

The Origo example shows how we may use *VamPeer* for messaging. It uses endpoint messages as well as Origo specific advertisements and documents.

Although the example has a nontrivial background, it serves as a good code repository where we can look at how we may implement some nifty tasks, for example how a private peer group is used or how we may create our own XML documents.

7 Results

We did several tests for the *VamPeer* library to show that the code works. There are some smaller examples, the Origo example, a benchmark and some unit tests. While we already mentioned all the examples, we would like to present the benchmark and the test results now.

7.1 Benchmark

Since *VamPeer* is going to be used for Origo which should be able to process many messages, we have to show that our library is able to handle much traffic and that it can process messages correctly.

Thus, we created a benchmark which is also shipped as an example. The benchmark consists of two peers, a master node and a slave node. The peers reside in the NPG and use discovery requests to find each other.

The master creates random strings and sends endpoint messages to the slave which uppercases the message contents and sends them back. The master checks at the end whether it has received all messages and whether the content was valid.

The master sends by default 1000 messages which is done in few seconds. *VamPeer* successfully passes this test as shown in listing 7.1.

```
2007/02/27 13:54:10.710 - FATAL - benchmark master: creating test data
2007/02/27 13:54:10.844 - FATAL - benchmark master: sending messages now,
    count: 1000
2007/02/27 13:54:14.533 - FATAL - benchmark master: message sending
    completed
2007/02/27 13:54:19.536 - FATAL - benchmark master finished successfully
2007/02/27 13:54:19.759 - FATAL - benchmark slave: gracefully stopped
```

Listing 7.1: Benchmark output

We see that the application works quite fast. To get such results, we have to finalize the code and turn the contract checking off. Else, it would last much longer. Especially the preconditions when parsing XML with the Gobo library take a lot of time.

7.2 Unit tests

To test some individual classes, we implemented some unit tests. There exist tests for the data classes that are used everywhere in the library. As it would be very cumbersome to have errors in these classes, we implemented tests for some advertisements, for the endpoint message and for all IDs.

The tests use the *getest* testing library by Gobo.

```
Test Summary for unit_tests

# PASSED:     44 tests
# Failed:     0 test
# Aborted:    0 test
# Total:      44 tests (1443 assertions)
```

Listing 7.2: Unit test results

7.3 Summary

In this master thesis, we built an Eiffel binding for JXTA. Although the implementation is far from being complete, the available parts are functioning and ready to use within Origo.

Speaking in JXTA terms, our *VamPeer* library provides the following services:

- The *endpoint service* handles incoming and outgoing messages and serves as a central point for other services. When messages are to be sent, the endpoint passes them to the correct transport module. Received messages from the network are passed from the transport modules to the endpoint service, which leads them to the appropriate service.

- We implemented a *TCP transport module* because it is simple and fast. It maintains a queue for outgoing messages and manages TCP connections with remote peers. It is able to generate and parse the wire message representation.

- Our *endpoint router* is very small and only provides the most essential feature: It resolves peer IDs to real destination endpoint addresses, namely for the TCP transport. It cannot send remote queries to get a peer's route information. The routes have to be available already when sending a message.

- Our *rendezvous service* implements just parts of the protocol. It only acts as an edge peer meaning that *VamPeer* is always in need of an external rendezvous server to be able to propagate messages within the peer group[1]. An edge peer is able to propagate messages to its rendezvous server.

- The *resolver service* is fully implemented in *VamPeer*. It provides a query-response system and may use the rendezvous to propagate a message.

[1]But we do supply such a rendezvous server. It uses the JXTA's reference implementation and is therefore written in Java.

- The *discovery service* serves as an advertisement storage and is used to query for advertisements, locally and remotely. Our discovery is also able to push SRDI messages to the rendezvous.

VamPeer is currently only able to run *one* peer group besides the WPG (which is only used for platform managing). But it is fully possible to replace the standard NPG with a private peer group.

Looking at the requirements in section 3.1, we have successfully fulfilled our task. With *VamPeer*, we are able to discover other peers and are able to communicate with them using an own infrastructure.

There is nevertheless the restriction that the user has to ensure the availability of the recipient peer's route information when sending a message. This can be easily done by once sending a peer discovery query and publish the results locally to the advertisement storage. As advertisements are subject to expiration, we will have to unset the lifetime before storing or to query again after a certain time.

We have well demonstrated how the library can be used by explaining the coarse API and by showing some examples: While the endpoint message example only focuses on the endpoint service, the discovery example describes how to search for remote peers. We also looked at a bigger example and argued how Origo could implement its messaging layer.

With these results, we release the first version of *VamPeer*.

8 Conclusions

The implementation of an Eiffel binding for JXTA was quite a challenging task.

As a JXTA novice, it was hard to get into the topic since many books and documentation about it are out of date.

Originally, we were full of expectations of the JXTA specification [Pro07] but we had to become aware of the fact that it specifies mainly the various messages' syntax. Detailed semantics is ceded to the implementation which makes it impossible to build a binding compatible to the reference implementation with the specification only. We therefore had to extensively look at the reference code to find out how the details really work.

We encountered also many uncertainties when we implemented the modules and tried to test our code because we did not see all problems during the design process. We encountered also a few bugs in the reference implementation while testing, which we could fix and send patches to the community. We are very happy that they all got applied rapidly.

While we now have a deeper knowledge in JXTA, we also learnt a lot about Eiffel software since we actively tried to stick to the Eiffel conventions. This was also a challenge since we always read the JXTA's Java code, which complies to other patterns than state of the art Eiffel code.

We are pleased to have started a new project *VamPeer*, which may grow in the future to become a mature and fully usable JXTA implementation.

8.1 Future work

We mentioned at several places before that our new library is incomplete and does not implement the full JXTA specifications. Although the current release can be used in Origo and in many others applications, it lacks of some essential features that would tremendously improve the *VamPeer*'s experience.

We really should focus first on a rendezvous server implementation because we then could eliminate the dependency to another JXTA implementation. This task involves working into the rendezvous' peer view protocol. While the rendezvous adaptions are a bigger part, it also involves to enhance the discovery service since this has to fulfill more tasks when the peer is a rendezvous server. We have to maintain an SRDI for all rendezvous clients for example.

Another important issue is the endpoint router completion. It should be able to resolve peer IDs also querying remotely for route advertisements. Another yet unavailable router feature is to forward traffic for other peers residing behind a firewall.

JXTA is actually also known for its support to bypass firewalls. The HTTP transport is surely an important contribution to this facility too. Such a transport would thus be nice.

Our thesis did not treat security issues. In JXTA, we generally own secure communication by using TLS channels. As there is no SSL/TLS library for Eiffel yet, it is not that easy to implement the `jxtatls` transport to *VamPeer*. But wrapping the SSL C library could lead to a neat TLS transport implementation.

Introducing support for several running peer groups should be an easy task as most services are ready for it. However, there may be some more challenges with the current address rewriting issue described in section 3.5.

When multiple peer groups are going to be allowed and secure communication is available, we may be interested to introduce the *membership service* which allows peers to gain access to a certain peer group only with a valid authorization.

The last missing service we would like to list here is the *pipe service*. Its idea is to support virtual channels to one or multiple peers. A simple implementation should not be that much work.

8.2 Acknowledgements

Without exaggerating, this thesis would certainly not have led to success if I had not get assistance by the JXTA community. The communication by means of mails and primarily IRC messages with other JXTA developers has really helped out with good advice. Therefore, I would like to thank Roger Karis and Mike Duigou (Sun Microsystems) for their great aid.

I like to mention Roger from Holland especially as he was also writing his master thesis in the JXTA field (writing about "Analysis of a peer-to-peer framework in μCRL"). We had some great hours in the JXTA's IRC channel sharing frustration and exultation.

I would also like to thank my supervisor Till Bay (the main driver behind Origo) who was always available to give advice and motivated me in my project.

The last six months were a demanding time but I had a good fellow sufferer with my office neighbour Samuele Lucchini who implemented a single sign-on solution for Origo. Thank you Samuele for the entertaining time!

Many thanks also to all my friends who did the proof-reading of this document: thank you Andrea, Mirjam, Nina and Philip!

Bibliography

[Bez07] Eric Bezault. *Gobo Eiffel Project.* http://gobosoft.com/eiffel/gobo/, January 2007. Version 3.5.

[BGKS02] Daniel Brookshier, Darren Govoni, Navaneeth Krishnan, and Juan Carlos Soto. *JXTA: Java P2P Programming.* Sams, Indianapolis, IN, USA, first edition, March 2002.

[Com06] Gerald Combs. *Ethereal: A Network Protocol Analyzer.* http://www.ethereal.com/, April 2006. Version 0.99.0.

[Eif06a] Eiffel Software. *EiffelNet.* http://www.eiffel.com/libraries/net.html, October 2006. Version 5.7.64493.

[Eif06b] Eiffel Software. *EiffelStudio IDE.* http://eiffelsoftware.origo.ethz.ch/, October 2006. Version 5.7.64493.

[Eif06c] Eiffel Software. *EiffelThread.* http://www.eiffel.com/libraries/threads.html, October 2006. Version 5.7.64493.

[FB96] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.* RFC 2046 (Draft Standard), November 1996. Updated by RFCs 2646, 3798.

[Goa07] Goanna. *Log4E.* https://svn.sourceforge.net/svnroot/goanna/trunk/log4e, January 2007. Revision 537.

[Hen06] Paul Hensgen. *Umbrello UML Modeller.* http://uml.sourceforge.net/, April 2006. Version 1.5.2.

[ISO04] ISO (International Organization for Standardization). *9834-8:2004 Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components.* ITU-T Recommendation X.667, September 2004.

[LMS05] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace.* RFC 4122 (Proposed Standard), July 2005.

[MKL02] Dejan S. Milojicic, Vana Kalogeraki, and Rajan Lukose. *Peer-to-Peer Computing.* http://www.hpl.hp.com/personal/Dejan_Milojicic/p2p_o.pdf, July 2002. HP Laboratories, Palo Alto, HPL-2002-57.

[MLK01] M. Murata, S. St. Laurent, and D. Kohn. *XML Media Types.* RFC 3023 (Proposed Standard), January 2001.

[Moa97] R. Moats. *URN Syntax.* RFC 2141 (Proposed Standard), May 1997.

[Pro] Project JXTA. *JXTA JSE Platform.* `http://platform.jxta.org/`.

[Pro06] Project JXTA. *SRDI: JXTA Shared Resource Distributed Index Design Plan.* `http://platform.jxta.org/java/srdi.html`, January 2006.

[Pro07] Project JXTA. *JXTA v2.0 Protocols Specification.* `http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html`, January 2007. Revision 2.5.2.

[TAD03] Bernard Traversat, Mohamed Abdelaziz, and Mike Duigou. *Project JXTA 2.0 Super-Peer Virtual Network.* `http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf`, May 2003. Sun Microsystems, Inc.

[Wil02] Brendon J. Wilson. *JXTA.* New Riders, Indianapolis, IN, USA, first edition, June 2002. `http://www.brendonwilson.com/projects/jxta-book/`.