UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Computer Science

Bachelor thesis

Implementation of Pointer-Push&Pull operation for maintenance of Random Graphs in Peer-to-Peer Networks

July 17, 2006

Student: Thomas Janson, tjanson@uni-paderborn.de

Advisor: Prof. Dr. Christian Schindelhauer

Abstract

In this bachelor thesis the implementation of a Peer-to-Peer Network with a random structure is presented. Each peer in this network is connected to a fixed number of other random peers. The Pointer-Push&Pull operation is used to keep the structure of the network random all the time. This operation is able to establish each arbitrary network structure with equal probability. The random structure with its expander property provides a robust connectivity to realize robust network backbones. Normally several random networks are used in an application for different contexts. For this purpose, a network communication model for Peer-to-Peer Networks is shown first which is able to handle many independent working peers at the same time and tries to solve the network communication between the peers in general. Finally a broadcast service is presented which is able to broadcast data in overlay networks like random networks.

Contents

List of Figures

List of Tables

1. Introduction

This work is focussed on the implementation of a Peer-to-Peer Network. The structure of this overlay network is random. The advantages of random networks are a small diameter, a robust connectivity even for small degrees, and the expansion property. Random networks are also successfully used in Gnutella and JXTA.

Chapter 2 describes the graph theory which is used in the implementation of the Peer-to-Peer Network. First the Pointer-Push&Pull operation, a distributed random digraph transformation [2], is presented. This operation is used in the implementation to establish random networks and maintain the connectivity of the networks. Furthermore, a technique for randomized rumor spreading [1] will be summarized, which is used for a broadcast service in overlay networks.

In chapter 3 the implementation of the Peer-to-Peer Network using the algorithms of chapter 2 is described. The first section of this chapter deals with a network communication model. In this model several participants like peers can use a network connection shared. The network layer is encapsulated from the application layer and participants exchange information in form of object oriented messages. The progress of the message exchange is asynchronous. Additionally, the communication model is extendable with a pipeline concept. It's possible to add extensions to this pipelines which can observe and modify the incoming and outgoing traffic.

In the next section, the implementation of a Pointer-Push&Pull Peer is presented. Pointer-Push&Pull Peers establish random overlay networks. The Pointer-Push&Pull operation is used to keep the network random all the time and to maintain the network connectivity. These peers base on the network communication model which allows applications to connect to several random networks in different contexts using only one network connection.

Finally the implementation of a broadcast service for an overlay network like the implemented random overlay network is shown. This service uses the technique of randomized rumor spreading described in chapter 2.

2. Graph theoretical consideration

2.1. The Pointer-Push&Pull operation

The Pointer-Push&Pull operation is an operation that transforms a labeled d-out-regular multi-digraph. The operation is introduced in [2].

The transformation of a graph is the transition of a graph to another one. The transformed graph has to contain the same set of nodes and has to keep to the same definitions like the degree. So the transformation of a graph is the modification of the edges set with restrictions to the definition of the graph.

A labeled d-out-regular multi-digraph is a directed graph with a regular out-degree d. So each node has exactly d edges to other nodes. Self-loops, which are edges from a node to itself, and multiple edges between the same nodes are allowed. Additionally, the edges are labeled with the effect that each edge has got its unique label, so that it can be distinguished between several edges between the same nodes. Furthermore, the graph is only weakly connected. This means that there has not to exist a directed path between all possible paris of nodes. A path between all nodes would only exit if the directed edges would be regarded as undirected.

The transformation of a graph $G = (V, E)$ in a Pointer-Push&Pull operation is defined as follows:

1. Choose a random node v_0 with $v_0 \in V$
2. Do a random walk starting at node v_0 with length 2. Choose in each step of the random walk an edge uniformly of the edges set of the node. The result is the path (v_0, v_1, v_2).
3. Transform the graph: Remove the edge (v_1, v_2) and add the edge (v_1, v_0). Remove the edge (v_0, v_1) and add the edge (v_0, v_2).

The transformation of the Pointer-Push&Pull operation is shown in figure 2.1.

The goal of this operation is to create random graphs. The advantages of this operation are that the connection of the graph is not interfered and it is possible to create each possible graph with the same probability. Random graphs provide a robust connectivity

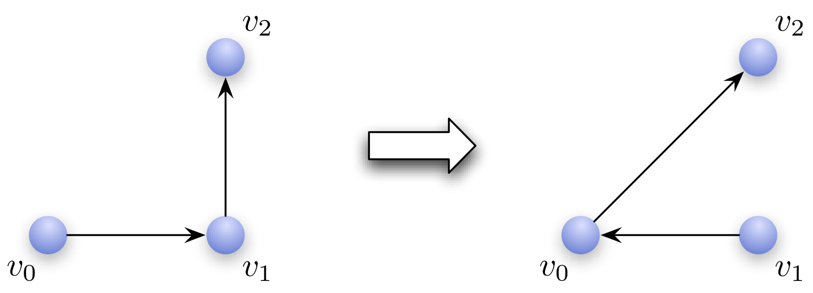


Figure 2.1.: The Pointer-Push&Pull operation

and an expansion, which addresses exactly the demands of Peer-to-Peer Networks. An implementation of the Pointer-Puhs&Pull operation in a Peer-toPeer-Network is shown in section 3.2.2.

Concurrent call of the operation

There is one case in the concurrent execution of the Pointer-Push&Pull operation, shown in figure 2.2, which leads to an incorrect state. In this example two operations are started

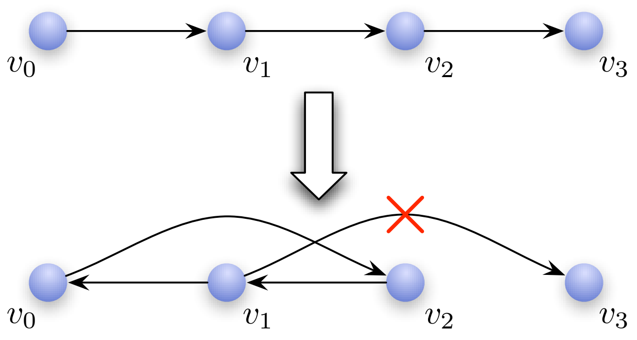


Figure 2.2.: Concurrent call of 2 Pointer-Push&Pull operations

at the same time with the random walk paths $P_0 = (v_0, v_1, v_2)$ and $P_1 = (v_1, v_2, v_3)$. If the first operation with P_0 would process the transformation and remove the edges (v_0, v_1), (v_1, v_2) and add the edges (v_0, v_2) and (v_1, v_0), then the second operation with P_1 would not be able to execute the transformation successfully because it could not remove the edge (v_1, v_2) twice. Therefore the edge (v_1, v_3) could not be added in the second operation because this would raise the out-degree of node v_1 by one.

A solution for this problem is to lock the edges on the random walk for other operations. So it is not possible that an edge is modified in the graph transformation of concurrent Pointer-Push&Pull operations several times.

2.2. Randomized Rumor Spreading

This section presents the theoretical approach of randomized rumor spreading which has been presented in [1]. This technique is used in the implementation for a broadcast service in an arbitrary overlay network.

The random phone call model is used for spreading the rumors. In this model n players are connected in a network. The process of rumor spreading is round based. In each round each player calls in parallel a random communication partner in order to exchange rumors. For the exchange of rumors between the called and the calling player there exist two possibilities:

- a rumor is *pushed*, if the calling player tells the randomly called other player a rumor.
- a rumor is *pulled*, if the called player tells the player, which has called, a rumor.

Here the push-pull-scheme is used, which is a combination of push and pull. The rumors are exchanged in both directions between the calling and the called player. In this scheme a rumor can be spread in $O(\ln n)$ rounds with $O(n \ln \ln n)$ transmissions of the rumor.

If a player is informed and spreads a rumor, it has to determine on its own when to stop spreading the rumor again. When a player stops too early and does not inform enough other players, it could happen that not all players are informed. If each player informs more players as needed, the according number of transmissions of a rumor between the players is higher as necessary, too. So this factor must be adjusted carefully. Here the median-counter algorithm is used for this task. The algorithm has to be executed for each single rumor by each player. In this algorithm a rumor has the state A, B-m, C or D. A rumor is in state A if the player has received the rumor the first time in the current round. In the states B-m and C the player spreads the rumor to other players. In state D a player does not spread the rumor anymore. If a player has a rumor in state D, it's possible that another player tells him the same rumor again. So the player has to keep the rumor in state D in order to prevent that it starts with the same rumor in state A again. After state D the rumor is deleted. The transitions of the states of the median-counter algorithm are shown in the diagram in figure 2.3 more precisely.

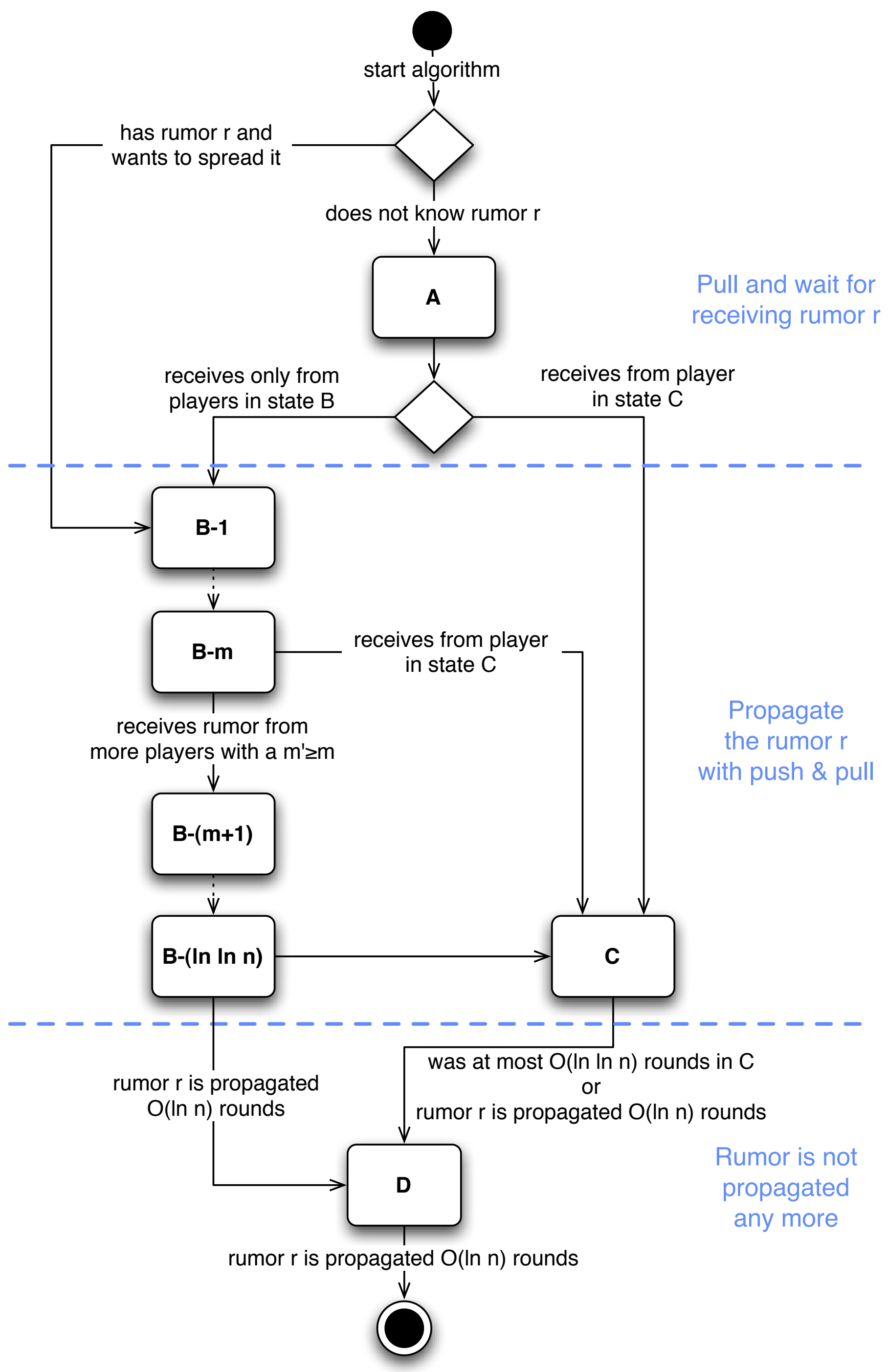

start algorithm
has rumor r and wants to spread it
does not know rumor r
A
Pull and wait for receiving rumor r
receives only from players in state B
receives from player in state C
B-1
B-m
receives from player in state C
receives rumor from more players with a m'≥m
Propagate the rumor r with push & pull
B-(m+1)
B-(ln ln n)
C
rumor r is propagated O(ln n) rounds
was at most O(ln ln n) rounds in C or rumor r is propagated O(ln n) rounds
D
Rumor is not propagated any more
rumor r is propagated O(ln n) rounds

Figure 2.3.: Sequence diagram of the median-counter algorithm.

3. Implementation

This chapter presents the implementation of the Peer-to-Peer Network. For the implementation *Java 1.5* is used. Additional information can be got from the sources or the *javadoc*. All Java specific names are marked with a writing without serifs (Example).

The description of the implementation is seperated in the following sections:

1. A network communication model for Peer-to-Peer Networks
2. A Pointer-Push&Pull Peer which participates in a random overlay network
3. A broadcast service which is able to spread rumors in an arbitrary overlay network

Figure 3.1 gives an overview of the usage[1] and the relations between the components.

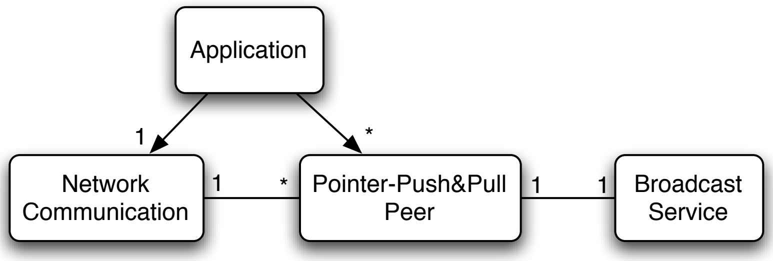


Figure 3.1.: Components of the implementation.

An application wants to use random networks. First it instantiates a network communication. Then it can create several Pointer-Push&Pull Peers. These peers use the network communication shared in order to communicate with other peers in the network. Each Pointer-Push&Pull Peer can join to a different random overlay network and is connected with other random Pointer-Push&Pull Peers in that network. A Pointer-Push&Pull Peer has a broadcast service. The application can broadcast information with this service in the connected random network. All other applications, which have a Pointer-Push&Pull Peer in the same random network, receive this information.

[1]An example application using random networks exists in p2p.randomnetwork.debug.ExampleApplication.

3.1. Basic framework for a Peer-to-Peer Network

Random overlay networks constitutes only a backbone for an entire Peer-to-Peer Network. This follows from the fact that random networks does not support a search functionality. It's only possible to process a random walk or to flood the random network for finding a certain element. So the search functionality can be established in a Peer-to-Peer Network which uses random networks only for a backbone on a lower tier. This structure consists of several partial networks with different types and structures. For this purpose a framework for Peer-to-Peer Networks is implemented which contains

- The general definition of a Peer-to-Peer Network
- The communication between peers
- Processing the protocols of peers

The individual Peer-to-Peer Networks can be created upon this framework.

3.1.1. Network communication model

The network communication model provides the shared use of a network connection for several participants. The participants are administrated with a local address. The participants exchange information in the form of messages. Messages are objects which are transformed to byte data for the network layer. For handling incoming messages a thread management is available. These threads forward the messages to the local receivers and the receivers can handle the messages in the same threads. The network traffic is passed through pipelines and additional features can be added to these pipelines in order to extend the model. Figure 3.2 illustrates this framework. All these features of the network communication model are combined in the class Communication. The structure of the model is shown in figure 3.3.

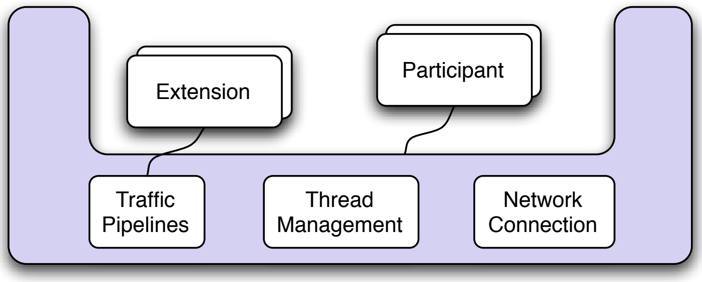


Figure 3.2.: Network Communication Framework.

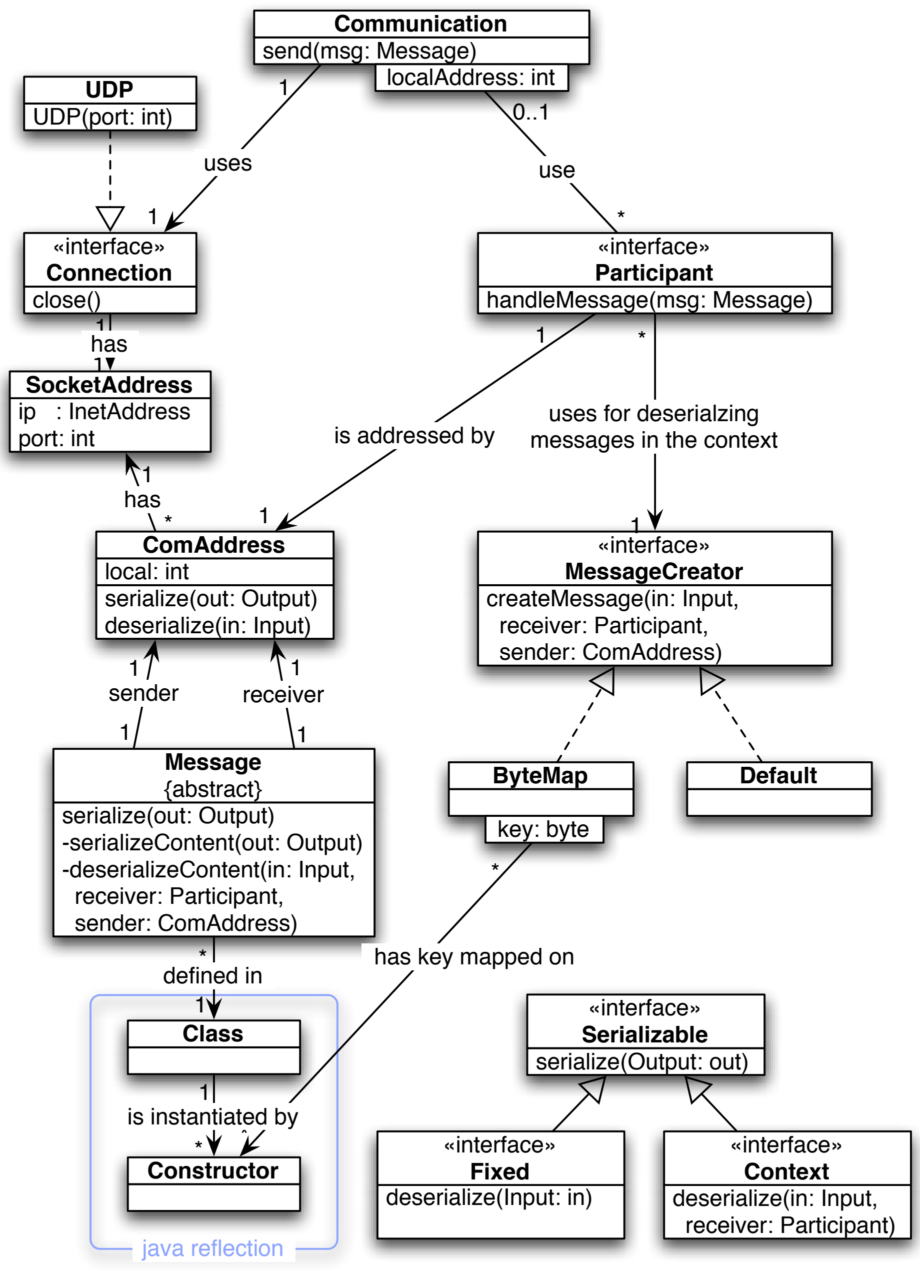


Figure 3.3.: Class diagram of the network communication model.

Local addressing

If several independent working peers are used in an application, it's necessary for the communication with other peers that each peer has a unique network address. The number of ports of a computer is often very limited. Several computers often share an internet connection and accordingly the port space, too. The user has to assign the port mapping in the router manually to the individual computers in the local network. Then a peer is able to act as a server and can receive messages from other peers which it has never contacted before. Besides most computers are firewalled. Under that security aspects the user wants to configure and release as few as possible ports.

Under this circumstances it's advantageous to use one single network socket shared for all independent working peers. A network socket can be addressed by the IP of the network connection of the computer and the port of the service within the computer. In order to distinguish the participant, which use a network socket shared, each participant gets another local address, that addresses the participants within the shared network socket uniquely.

So in this network communication model each participant has a ComLocation which is composed of

- IP address
- port
- local address

The local addresses are assigned to the participants during the registration in the Communication. A coincidental still free local address is usually assigned to the participants. But it's also possible to use a fixed address. Participants with a fixed address can be addressed with only the IP and port. A fixed local address can also be reserved as a precaution, that the address is not assigned accidentally to another participant.

Information exchange based on messages

This network communication model is designed for small information exchanges. In Peer-to-Peer Networks only small messages are sent for maintaining the overlay network structure or executing the lookup operation.

The communication participants like peers exchange information in form of messages. The messages are objects and have the base type Message. A Message object contains the sender address, the receiver address, and the content which should be transmitted. The sender address is needed in most cases for an interacting protocol and for the assignment to an operation which is executed cooperated. The receiver address is needed to send the message to the receiver. The content of a message can be defined in a special class derived from Message. In the definition of the content can be distinguished between the object layer and the raw data layer. In the Message-object the content can be composed and described object-oriented how it's used in the protocol of the communication participants. If a message is sent via the network, it's serialized to a byte array and vice versa on the receiver's side. So the protocols in the communication participants like peers are encapsulated from the direct network communication and can be defined on a more abstract level. The course of communicating works as follows:

- Sender:
  1. Participant:
     a) Compose the content in a special Message-object, which should be sent
     b) Call the send-method in the Communication
  2. Communication:
     a) Serialize the Message-object to a byte array
     b) Send the byte data via the network to the receiver
- Receiver:
  1. Communication
     a) Receive the byte data of a single message
     b) Lookup the Participant-object, which is the receiver with the local address
     c) Associate the byte data with a Message-type in the context of the receiver, which defines the structure of the data. Create an instance of that Message-type
     d) Deserialize the byte data to the Message-object in the context of the receiver
     e) Call the handleMessage-method in the receiver-object
  2. Participant
     a) Process the received Message-object

The serialization process is the transformation of a Message-object to the byte data which can be sent via the network. In this model the way how to serialize is defined in the Message-type. Standardly the standard object serialization of Java is used. In this ObjectOutput stream an object is serialized in the general form. First the name of the class of an object with the whole package information is written, followed by the serialVersionID which is used to avoid that two different versions of the same class are used. Then for each attribute the name and the data is written. Referenced objects are added on the same way. The references between the objects are represented with ID's. So if one object is referenced several times, it's only written to the stream exactly one time.

So the standard serialization has a huge overhead of data sizes. The first element in the serialization is the class-type which can be equated with the type of the Message. A communication participant like a peer has only a small amount of different message types in its protocols. So if it is known which type of service like a peer receives a message, then the type can be represented in the context of the receiver with a smaller-sized value like only a byte. A byte is the smallest unit for the serialized byte data. Instead of writing the name of an attribute for the identification, it is also possible to arrange the attributes in a fixed order. Mostly nested objects within a Message-object have a flat hierarchy and the type of the objects is not various with generalization. So it's not always necessary to describe the object type in the byte array. An example is an IP-address, which is represented in Java as an object. This objects can be also treated as an 4 byte attribute in the serialization.

So summarized the standard object serialization has the advantage, that it is very easy for the programmer to define the objects which should be sent in the network. This contains particularly general not fixed types, which are replaced in the instance situation by interface realizations or derived classes. So it has not exactly to be known what data has to be transmitted. But the general serialization has the disadvantage that the size of the data is not optimal. So it is always a consideration between a small network traffic and an easy implementation. In the context of Peer-to-Peer networks often small messages are exchanged in order to maintain the overlay network and in this context this messages should be as small as possible in order to minimize the base traffic of the network. As result it can be chosen in the framework between the standard serialization and a user defined serialization. For this purpose a Message-class has to be derived from the interface Communication.Serializable if the way of the serialization is explicit defined by the programmer. Else the standard serialization of Java is used.

It has to be clear in the deserialization how to transform the byte data to a Message-object again. In the deserialization process the Communication determines the receiver of the message first. The receiver provides a so called MessageCreator-object which defines how to deserialize a message in the context of the receiver. The derived class

MessageCreator.Default deserializes a message with the standard java deserialization with a ObjectInput stream. For the user defined deserialization a MessageCreator with the derived type MessageCreator.ByteMap is implemented. This type can be used if it's possible to represent all possible message types with a single byte. It's sufficient that each type of communication participants instantiate one MessageCreator.ByteMap, because all instances of the same receiver type will act on the same way. In such a MessageCreator.ByteMap the map from a single byte to the class type of the message can be defined. This is possible in Java with the *java reflection*. In the java reflection an object of the class type Class can be got from each java class. The Class-object contains the whole class architecture description with all attributes, functions and constructors in a general form. So it's possible to access all elements of a java object in a general way. So more precisely a MessageCreator.ByteMap contains a map from a byte to a Contructor-object of the java reflection of the corresponding Message class type. A MessageCreator.ByteMap reads in the deserialization process the first byte of the byte array of a message and determines the mapped Constructor. With that Constructor it can instantiate an object of the individual Message type. Then it calls the deserialize-method in that instance with the input stream of the byte array as parameter. In that object-method the content is read from the stream and set in the Message-object.

The composition of the message data follows to the class architecture of a derived Message class. Dependent on that definition the individual serialization of the object-oriented content can be described in the object method serializeContent(). This method has a stream of a byte array as parameter. The content of the message can be written to this stream. The serialization takes place in the Communication which is described more precisely in the next section. The deserialization is the transformation from the raw data of a message to the object-oriented Message-object. First an object of the special Message class is instantiated. Then the object method deserialize() is called in the Message-object with a input stream of the byte data as parameter. The content can be read from the input stream and set in the object similar to the execution in the serialization. A further parameter is the reference to the communication participant. This participant receives the message when it's completely created. This reference can be used if some elements of the content are various and the participant has the information how to deserialize the element. This technique is described more precisely in the section 3.1.2.

The local addresses of the sender and receiver of the message are added as further header like the IP header before the content in the raw data which is sent via the network.

Network Connection

In this model the network connection is encapsulated from the application layer. It's bounded to the Communication as an object which has to implement the interface Connection. A Connection has to contain an incoming and an outgoing channel for the network. The incoming channel receives incoming byte arrays of messages. The outgoing channel is used to send byte arrays to to a socket address via the network. Both channels are connected in the Communication. The participants of the Communication have no direct access to this network connection object. They call the send-operation in the Communication for sending a message and the Communication forwards incoming messages to the receiver.

The default implemented connection is based on the *UDP* protocol. UDP stands for User Datagram Protocol. This protocol gives a direct access to the datagram service of the IP layer. In contrast to TCP, which provides a connection-oriented byte stream, UDP offers only the delivery of single datagram packets. That adapts exactly to the communication model with small single messages. A datagram packet consists only of the address, the ports, a checksum and the data, which has a variable length. The maximum size for a single network packet is 1492 byte. If the content of a single datagram package is larger, the package is splitted into several packets whereas only the first one contains UDP header. So it might be useful to take care about that limit and to utilise the full capacity of single packets.

Each packet is individually addressed and routed. The delivery of the sent packets is not guaranteed (no flow control). It's also not guaranteed that the packets are received in the same order in which they have been sent.
But all those restrictions, which would be supported in the TCP protocol, are not absolutly needed in the case of a Peer-to-Peer Network. In the communication of a Peer-to-Peer Network only small messages are exchanged between the peers. This messages contain often a request which expects a reply. So the acknowledge of the successful delivery of the packet is returned automatically with the reply for the request.
An advantage of UDP is in contrast to TCP that the bandwidth is better used by sending small packages and the overhead is minimal.
A socket with the TCP protocol establishes a fixed connection with another socket. So a connection has to be established to each different peer in the network for the communication. That would raise the costs in a high dynamic network because a new connection has to be established to each new peer of the permanently changing set of neighbors. The UDP protocol is easier to handle: One DatagramSocket is used for sending and receiving the messages of all arbitrary other peers and no connection is needed to set up.

For the outgoing and incoming channel the same socket with the same port is used. This is necessary because the Communication acts as a server and the sender address of a request has to be used as the receiver for a reply. It wouldn't be possible to add the server port to the content of a network packet because that port would not be translated with NAT, if the computer is behind a router.

Design of the network channels as pipelines

The incoming and outgoing channel of a Communication are designed as pipelines. Several elements can be added to these pipelines which handle the incoming and outgoing messages. Figure 3.3 illustrates that structure. In this way extensions can be added to the Communication. The traffic is forwarded through each element of the pipelines. So the elements can handle each message and modify the traffic or provide additional services for the single messages. Thereby both pipelines are segmented in the object layer and the raw-data layer. The pipeline elements in the object layer process the messages as Message-objects. The elements in the raw-data layer process the messages as byte arrays. Between the object layer and the raw-data layer two elements convert the traffic between the layers and serialialize and accordingly deserialize the messages.

In this structure the communication participants are connected to the beginning of the object layer of both pipelines. They call the send-operation in order to send a Message. The Message is injected in the outgoing pipeline. Messages which reach the end of the incoming pipeline are forwarded to the receiver. At the end of the raw-data layer the network connection is bounded. Outgoing messages leave the Communication to the network. Received messages from the network enter the Communication.

One general service is an acknowledgement service. This service guarantees that each sent message has been received successfully. The service is added to the object layer of the incoming and outgoing pipeline of a Communication. A message which should be acknowledged is added to a special container with the class type MessageAcknowledgedContainer. This container contains additionally an ID, which identifies the message uniquely in the Communication. The outgoing pipeline element buffers all these messages. It adds them to a scheduler as a TransmitErrorMsg with a timeout. The timeout indicates when the acknowledgement of the message has failed. A scheduler is a simple utitility which allows to send a message at a certain time with the linked Communication. The container with the original message and the ID is passed on normally in the outgoing pipeline and is sent. If a Communication receives such a message, it replies with an acknowledgment message with the type AcknowledgementMsg. This acknowledgment contains the ID. If a Communication receives such an acknowledgement message in time, the buffered transmit error message is removed from the scheduler. Else the transmit error message with the type TransmitErrorMsg is sent to the sender of the real message. So the communication participant can react to the transmit error.

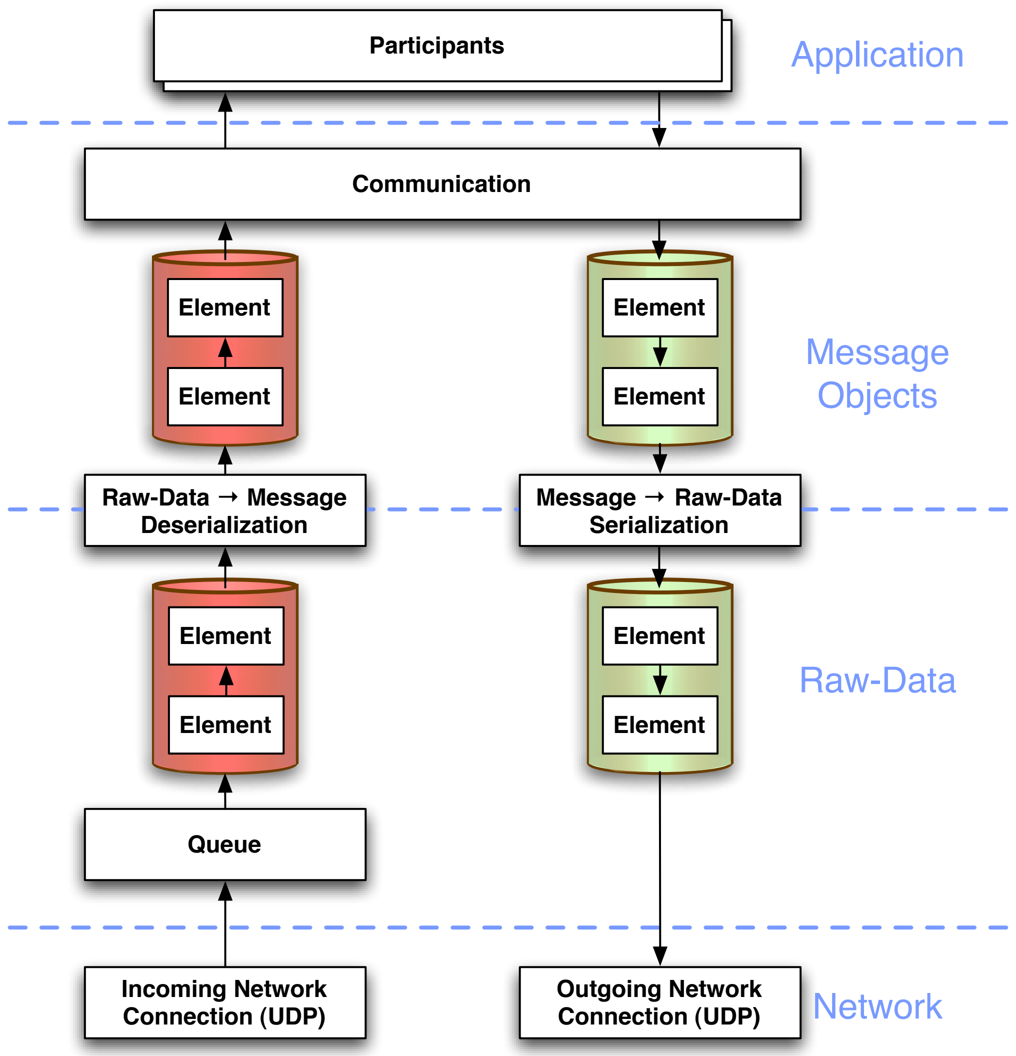


Figure 3.4.: Pipe architecture of the abstract network communication model.

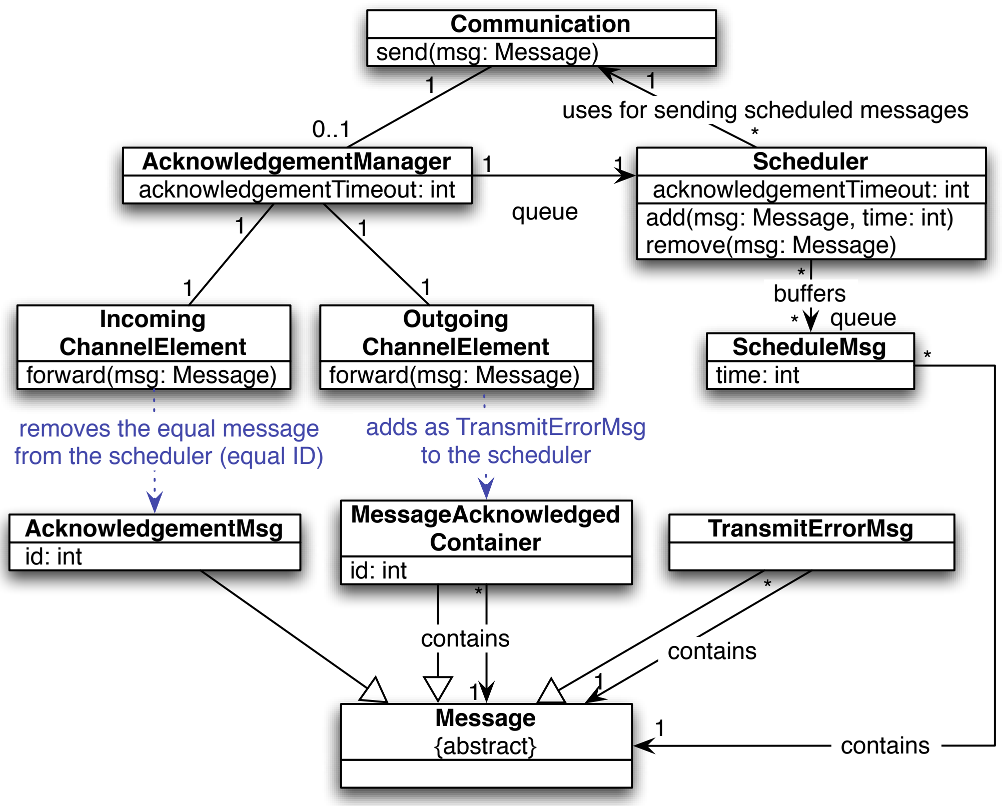


Figure 3.5.: An acknowledgement manager as pipeline element of a Communication.

Another useful service is to retour local messages which have the same Communication as receiver. This service is added to the object layer of the outgoing pipeline. It checks simply the receiver address and redirects all local messages to the entrance of the object layer of the incoming pipeline. A local message has the address of the Connection of the Communication as receiver. In this case Message-objects have not to be serialized and to send via the network. This is mostly of interest in the simulation of a Peer-to-Peer Network where several peers can be tested in one Communication with high performance.

A further interesting part for testing the robustness of a overlay network is of course to manipulate the network connectivity. For this purpose, two elements can be added to the incoming and outgoing pipeline. This elements dispose of some messages or forward messages with a random delay.

Thread management

This model provides an asynchronous communication between participants. Participants can send a request message and get an asynchronous reply message. The asynchronous communication provides parallelism. A Communication has a queue where all incoming messages are buffered. The incoming channel of the network connection passes the the messages as byte arrays to that queue (Figure 3.4). Several worker threads handle the buffered messages in the queue. The number of threads can be adjusted in the method Communication.setNumberOfMessageHandlers(int). A worker thread removes an available message from the queue and process the flow of this message through the incoming pipeline. Finally it delivers the message to the local receiver. The receiver is a participant of the Communication. The participant can handle the received message in the same thread. It can react on the received message and send a reply for example. So a participant needs only own threads for executing own tasks.

The size of the queue for incoming messages is limited in order to prevent a buffer overrun. If the quantity of inquiries cannot be mastered, there is no other possibility as deleting some messages. Here the oldest messages are deleted and not handled. It would also be conceivable to delete the messages dependent on a priority.

A participant calls the method Communication.send(Message) in order to send a message. This operation is executed in the same thread until the message is sent and leaves the Communication. If the bandwidth of the outgoing network connection is too small for the produced traffic of the participants, the time for sending raises. So the participant can measure the utilization of the Communication and is able to adapt accordingly it's behavior.

Service for resolving the own public address

A further general problem is that a network service does not know its own public address in the internet consisting of the IP and the port. If a computer uses a router as gateway to the internet, it's in a subnetwork and shares the internet connection with other computers in the same subnetwork. In this case each computer has got an address in the local network and the router has got one unique address in the internet. If a computer sends a packet out from the local network into the internet, the router translates the address in the packet header and replaces the local address with the public address. So the receiver can send a response to that public address and the router forward the response back to local computer. In this step the public address is replaced by the local address again. As result a service does not know if an address, which it has to another service, is really another service or it is its own address. This circumstance will not disturb the functionality of the service but it would save some performance and a service does not have to send data to itself.

For this purpose a service has been implemented which determines the public address of a Communication. This service is a Participant of a Communication. It has a fixed local address in the Communication. So the socket address of the Connection of a Communication combined with that fixed local address is sufficient to address the service. The simple idea of this service is that several of this services provide the public address each other.

It's often the case that the internet connection has got a dynamic IP address which is renewed recurrently for example every 24 hours. So a service has to refresh and determine its own address in an interval that it doesn't become outdated. But the time of the replacement of the address is not known. So it cannot be guaranteed for each point in time that the own address is correct and the own address should only be used as optimization.

In the protocol of the service each service sends in an interval a request message to another random service and gets a reply back. Both messages contain the socket address of the other service in the payload of the packet. This address remains unchanged and is not translated with NAT. So in one interaction both services can provide each other its own address. The service has no own routing table and does not know which other service it can contact. So it needs another service which is able to provide addresses of other Communications. This can be some arbitrary other service like a peer which has to implement the interface CommunicationAddressProvider. This interface contains a function which results a random socket address of another Communication. It's additionally possible to register other objects which are informed when the socket address is renewed in the Communication. This objects have to implement the interface AddressChangesListener.

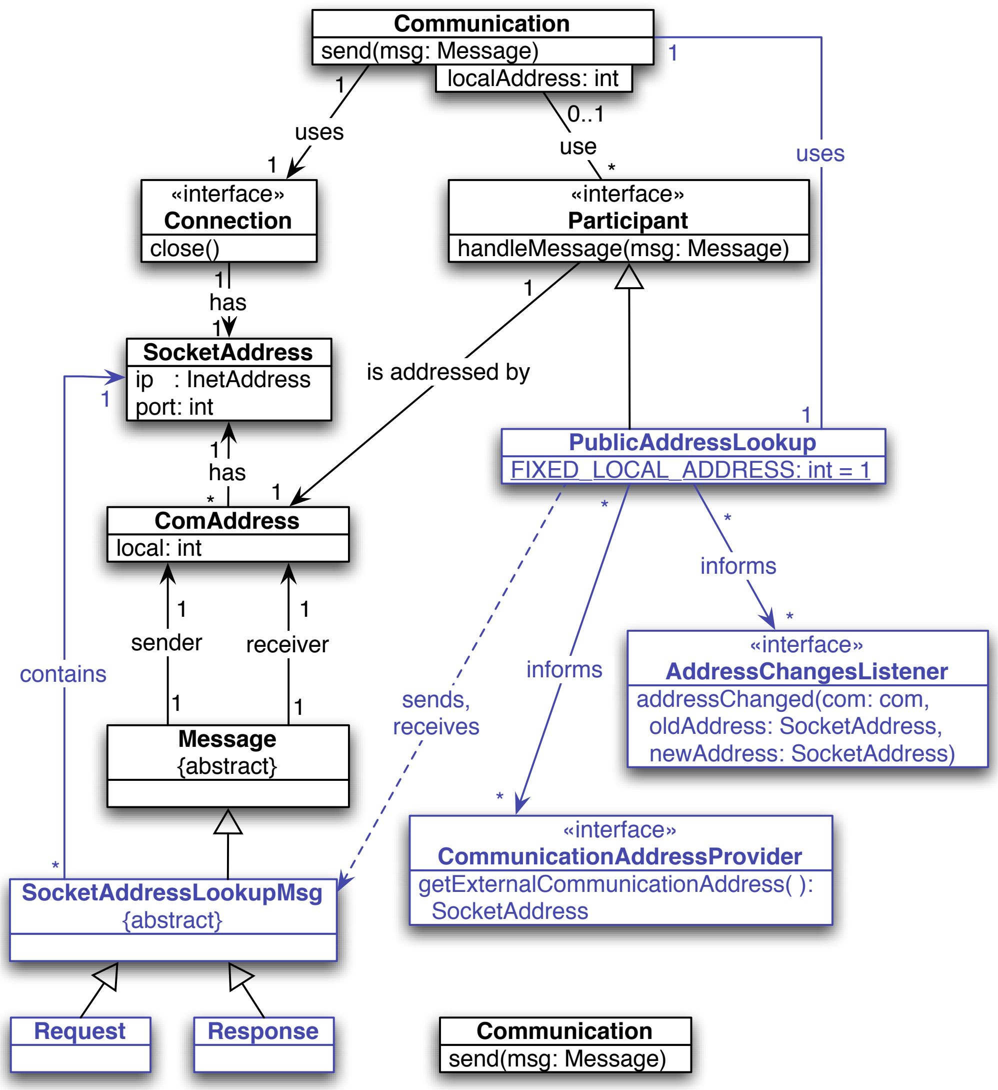


Figure 3.6.: A service which resolves the public socket address of a Communication.

3.1.2. Peer interface based on the network communication model

This section describes an interface for peers which are created upon the network communication model described in the preceding section. This interface allows also a general integration of Peer-to-Peer Networks in applications. Figure 3.7 shows the interface.

In this model an application instantiates a peer with a factory with the base type PeerFactory. A concrete factory, which is derived from this base type, is able to produce peers with the type Peer for the individual used Peer-to-Peer Network.

A Peer has to implement the following common operations:

create The peer creates a new overlay network. The properties of the network are defined by this peer. All other peers have to keep to this initial properties.

join The peer joins to a still existing overlay network. The parameter of this operation is the address of a known peer of that network. The peer can contact the other peer and receive all information which it needs to join to the network. This includes for example the individual properties of the overlay network.

leave The peer is currently connected to a network and leaves it.

lookup A lookup for a peer which is responsible for a certain ID in the Peer-to-Peer network is executed. The passed parameter of this function is the searched ID. The type of the ID depends on the individual type of the Peer-to-Peer network. The result of the function is the address of the peer that is responsible for the ID. The type of this address is PeerAddress.

A peer is a special Participant of a Communication and uses the Communication to exchange messages with other peers in the same overlay network. Each peer has a routing table with addresses of some other peers for the connection to a Peer-to-Peer Network. The information and addresses of a peer are bundled in a PeerAddress which consists of the following 3 elements:

- The network address
- The address or ID in the overlay network
- Some information of the application which uses the peer

With this composition of all information about another peer, the routing table of a peer can be created in simple data structures. The network address is needed for communicating with other peers. The address of a peer is in this framework a ComAddress which has been defined in the section 3.1.1.
The overlay network address or ID is needed for executing the lookup operation and for arranging the routing table. This is for example the ID in a distributed hash table.

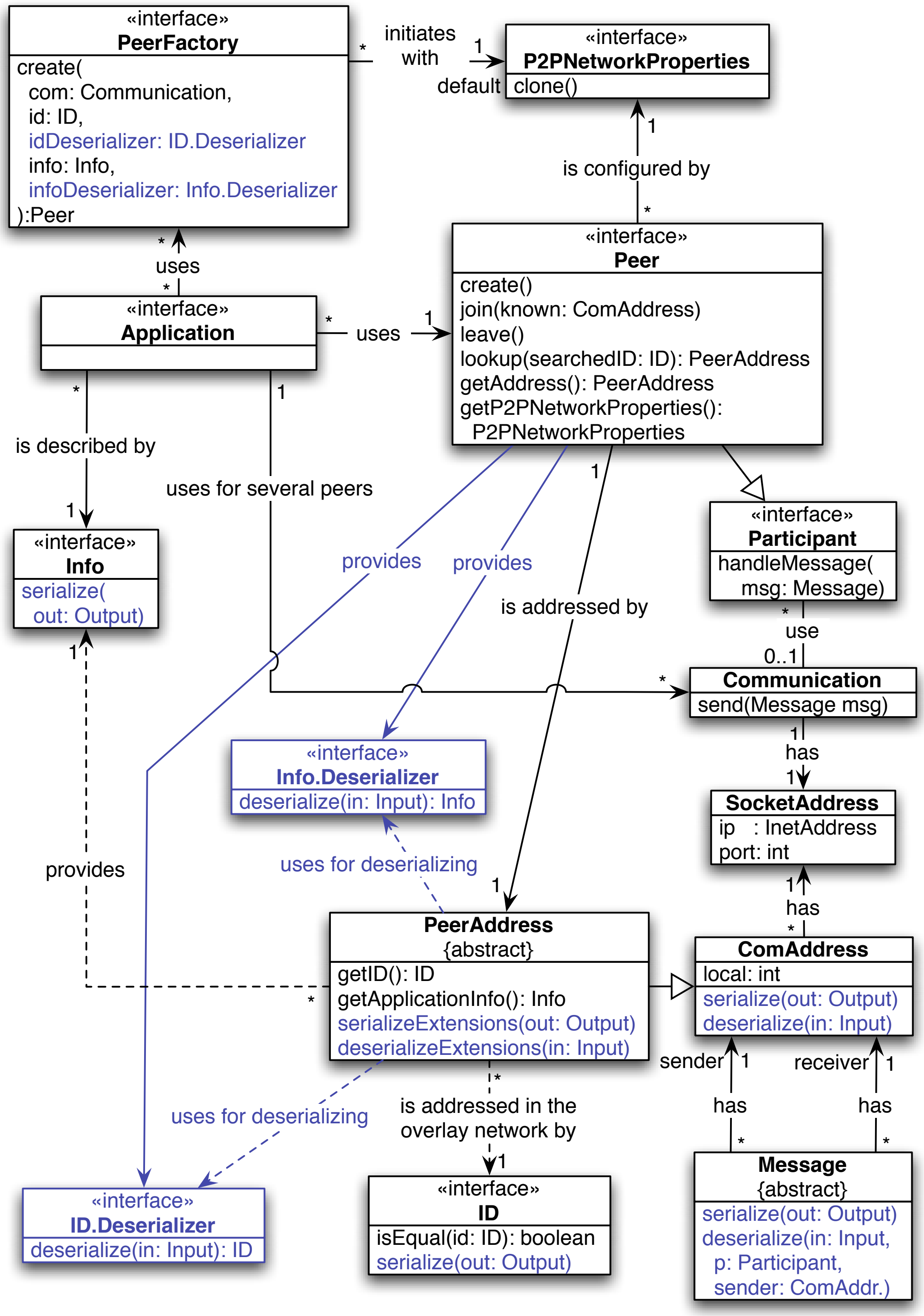


Figure 3.7.: The *Peer* interface based on the abstract network communication model.

The result of the lookup-operation is a PeerAddress. The application, which receives the result, wants rather communicate with the application behind the found peer. So a PeerAddress contains additional information about the application which uses the peer. This information is various and defined by the application and could be for example the port of the network connection of the application. So the requesting application is able to communicate directly with the other application of the found peer.

PeerAddress's are exchanged between the peers of a Peer-to-Peer network. So the addresses have to be serialized and transmitted via the network. The serialization of the network address is defined in the method ComAddress.serialize(). In this method the network address is written to the output stream and the method serializeExtensions() is additionally called. The derived class PeerAddress overrides this method. The extensions with the ID in the overlay network and application information are additionally written to the output stream. The method ComAddress.deserialize() is called for deserializing an address. The network address is read from the input stream and the method deserializeExtensions() is additionally called. This method is overridden by the PeerAddress and the extensions of the PeerAddress are read from the stream. The type of the application information and possibly the type of the ID in the overlay network is various. So it has to be defined in the Peer with a Info.Deserializer and ID.Deserializer object how to deserialize that data. Both elements are able to read the data from the input stream and to return the according object.

Restrictions relating to NAT

A peer does not always directly know its own address. If the peer runs on a computer that uses a router as gateway to the internet, the computer has an address in the local network and the router has a unique IP address in the internet. If a peers sends a packet, the router replaces the local address in the packet header by the public address with NAT. The port can also be translated.

In this background a peer is not allowed to serialize its own IP and port in the data of a message. If the own address of the local network would be transmitted in the payload of a packet, it would not be translated with NAT. And the local address can only be used in the local network. As result the peer cannot add its own address to the routing table. If other peers would send a request, it cannot reply with that entries. So the first peer in a network has to start with an empty routing table instead of filling the routing table with the own address according to the definition of the network structure. If a second peer joins the network and demands for addresses of the network, both peers can transmit one another the own addresses and both can initiate their routing tables. With this strategy it's not necessary to serialize the own IP or port in the user data of a message.

3.2. Pointer-Push&Pull Peer

A Pointer-Push&Pull Peer is a peer of a random network which uses the Pointer-Push&Pull operation to keep the network in random and to maintain the network connectivity. The Pointer-Push&Pull operation is introduced in section 2.1 .

The peer is implemented in the class PointerPushPullPeer. According to the definition of the Pointer-Push&Pull operation this peers have a multi-set as routing table with the size of the regular out-degree of the network. The routing table contains addresses of neighbors in the random network. Multi-set means that it's possible that the same neighbor can be contained in the routing table several times. Additional abilities of this peer are a lookup operation for a peer with a specific ID and the usage of a broadcast service. The architecture of the broadcast service is more exactly described in the next section 3.3. Table 3.1 gives a survey of all properties of a random network. All peers of the same random network have to keep to the same properties. This properties are implemented in the class PointerPushPullProperties.

Table 3.1.: The parameters of a Pointer-Push&Pull Peer.

Category	**Parameter**	**Unit**	**Size [Byte]**
Degree	Regular Degree	Integer (number)	4
	Old neighbors	Integer	4
Pointer-Push&Pull	Interval	Integer (milliseconds)	4
	Timeout	Integer (milliseconds)	4
Random Walk	Number per hop	Integer (number)	4
	Timeout	Integer (milliseconds)	4
Lookup	Maximum hops	Integer (number)	4
	Timeout	Integer (milliseconds)	4
Bearer service	Lost connection timeout	Integer (milliseconds)	4
Broadcast service	Interval	Integer (milliseconds)	4

The Pointer-Push&Pull Peers use the network communication model described in section 3.1.1 with a Communication for exchanging messages with other peers. This type of a peer is only able to send and process a small amount of different message types. These types are shown in table 3.2. All these message types are individual serialized and each type is mapped to a 1 byte key. A MessageCreator.ByteMap is used in the deserialization process in order to map the byte key to a message type.

The following subsections deals with the algorithms for establishing the random overlay network and what is executed in the main operations create, join, leave, and lookup.

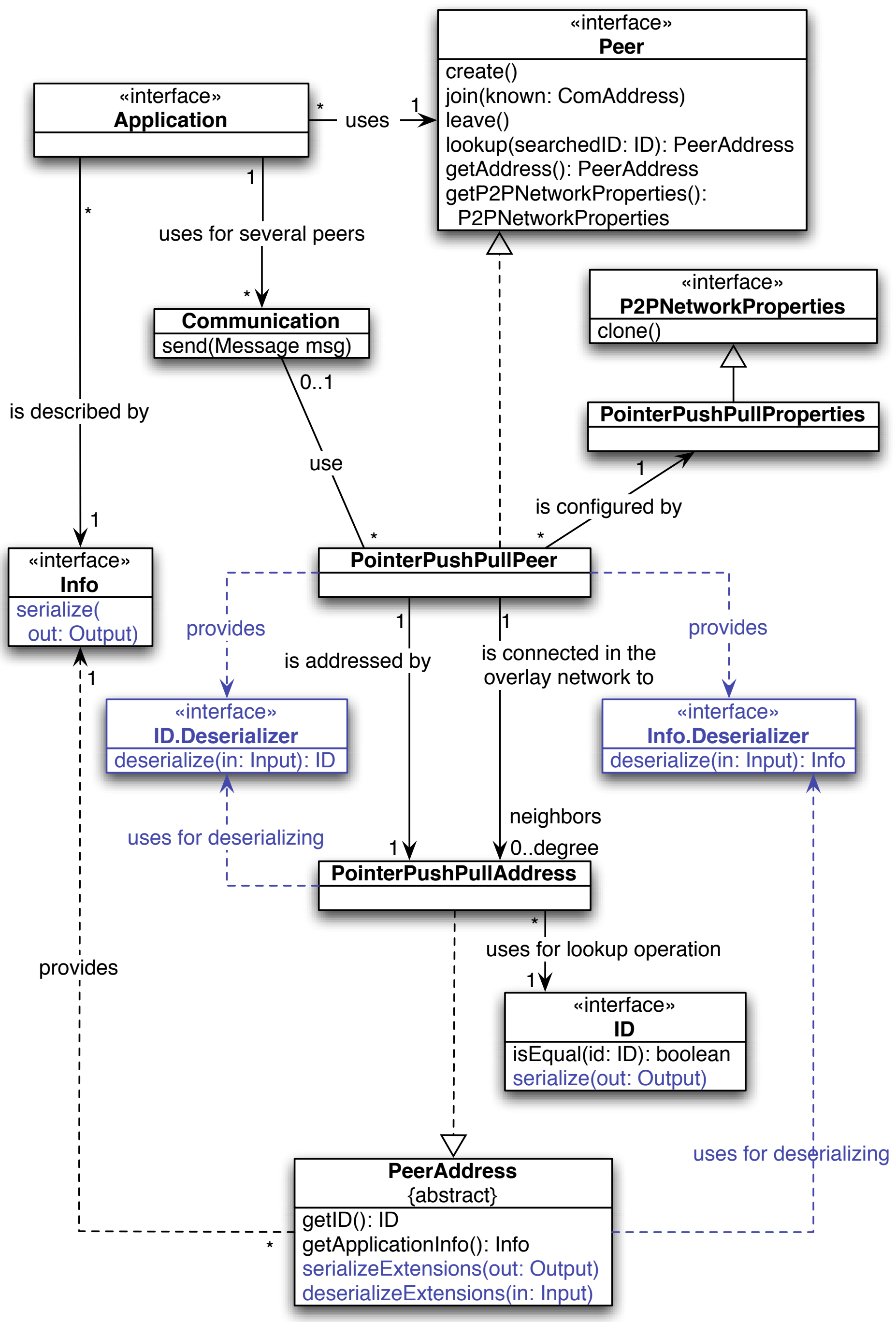


Figure 3.8.: Class diagram of the structure and use of a Pointer-Push&Pull Peer.

Table 3.2.: The different message types used by a PointerPushPullPeer.

Key	**Mapped message type**
0x00	Lookup request
0x01	Lookup response
0x02	Pointer-Push&Pull request
0x03	Pointer-Push&Pull response
0x04	Random walk request
0x05	Random walk response
0x06	Random walk join request
0x07	Random walk join response
0x08	Broadcast pull request
0x09	Broadcast pull response
0x0A	Broadcast push

3.2.1. Random walk operation

A random walk is path through a graph. On each step of the path the next node is randomly chosen with same probability.

A random walk operation is used here to collect addresses of peers in the network. This operation is used when a peer has to few links in its routing table. Then it has to fill up the routing table in order to have the regular degree. This is the case if the peer wants to join to a network or if it has dead links in its routing table and has to repair the routing table.

The random walk is controlled by the peer which has initiated the operation. In the first step the peer sends a request message to a random neighbor of its own routing table. If a peer receives such a request message, it replies with a random neighbor of it's routing table. So the peer, which executes this operation, can use that address for the next step of the random walk.

On each step of the random walk a defined number of random links are collected and sent back to the peer which has initiated the operation. So that peer is able to fill up it's own routing table until the regular degree of the random network is reached.

If the first step of the random walk is processed in the join operation, the joining peer has not the parameters of the network. Then it sends a request message with a special type for the join-operation on the first step of the random walk. In this case the peer

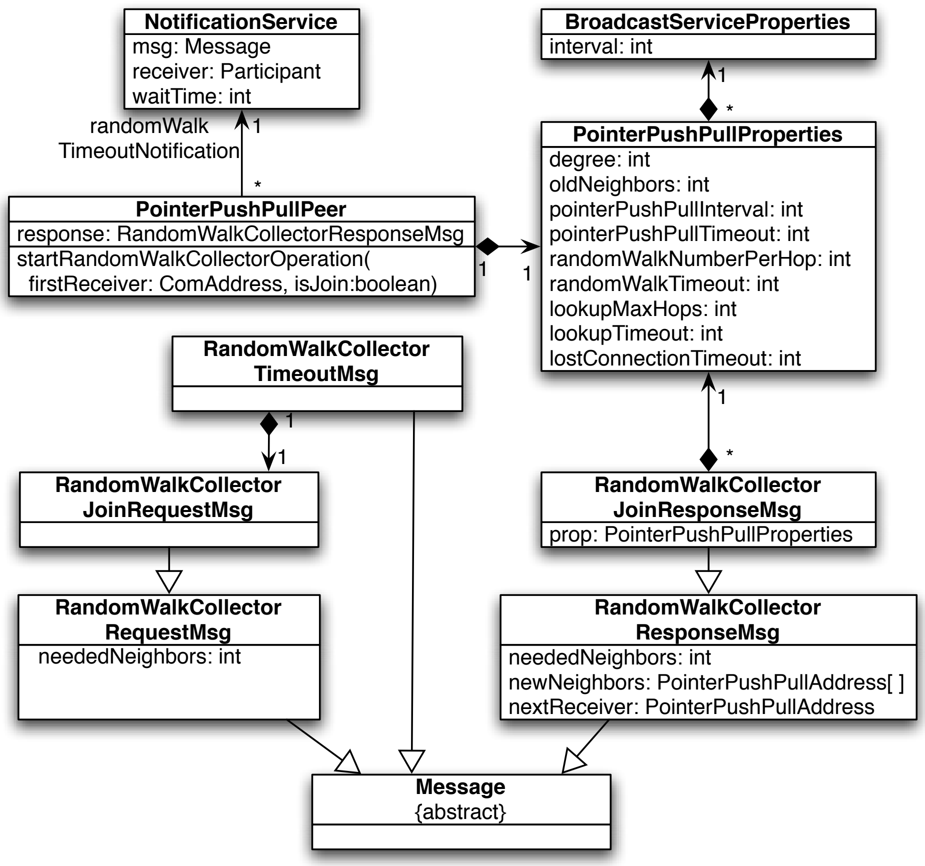

NotificationService
msg: Message
receiver: Participant
waitTime: int
randomWalk TimeoutNotification
1
*
BroadcastServiceProperties
interval: int
1
*
PointerPushPullProperties
degree: int
oldNeighbors: int
pointerPushPullInterval: int
pointerPushPullTimeout: int
randomWalkNumberPerHop: int
randomWalkTimeout: int
lookupMaxHops: int
lookupTimeout: int
lostConnectionTimeout: int
PointerPushPullPeer
response: RandomWalkCollectorResponseMsg
startRandomWalkCollectorOperation(firstReceiver: ComAddress, isJoin:boolean)
1
1
RandomWalkCollector TimeoutMsg
1
1
RandomWalkCollector JoinRequestMsg
1
*
RandomWalkCollector JoinResponseMsg
prop: PointerPushPullProperties
RandomWalkCollector RequestMsg
neededNeighbors: int
RandomWalkCollector ResponseMsg
neededNeighbors: int
newNeighbors: PointerPushPullAddress[ ]
nextReceiver: PointerPushPullAddress
Message
{abstract}

Figure 3.9.: Class diagram of the structure for the random walk operation.

which replies this message, defines on its own how many new addresses it has to send back and sends additionally the network parameters to the joining peer.

In order to handle the case that a request is not replied, a peer starts a notification service. This notification service is able to send a message time-delayed back to the peer. So the peer sends with this service a message delayed with the timeout for the random walk operation. If the reply is received in time, this notification service is simply finalized. But if the timeout is expired, the peer knows that the other peer will not reply any more. In this case it tries to continue the random walk with another random link.

3.2.2. Pointer-Push&Pull operation

The Pointer-Push&Pull operation is used to keep the overlay network random and to maintain the network connectivity. The class diagram in figure 3.10 shows all classes of the implementation which are involved in this operation.

The operation is executed on each peer in rounds. A Timer object is used to initiate the recurrent call of the operation. A Timer is an extended thread which is able to send recurrently a message in a fixed interval to a Participant like the PointerPushPullPeer. The used message has the type PointerPushPullStartMsg. Whenever a peer receives this message, it initiates the Pointer-Push&Pull algorithm. In order to make it possible that each possible random network can be produced with equal probability, a peer does not execute the operation in each round. The number of rounds, which a peer has to wait for the next execution, is determined in a Markov chain. So after each execution of the operation the next interval is determined in a Markov chain. In this operation a loop is iterated and in each iteration stopped with probability 0.5. The number of iterations is at the same time the number of rounds when the peer will initiate the next Pointer-Push&Pull operation. The interval in the Timer-object is adjusted accordingly.

If a peer starts the Pointer-Push&Pull operation, it first chooses an arbitrary link of its routing table neighbors with equal probability. This link is locked and moved from the list neighbors to a data structure with the type LockedAddresses. This avoids that another concurrent operation can be executed with the same link. It sends a request message with the type PointerPushPullRequestMsg to that random neighbor.

If a peer receives such a request message, it chooses a random neighbor of its own routing table. If this address is locked and is in the data structure LockedAddresses, the operation has to be canceled and a response message with no new neighbor is sent back. Else if the randomly chosen address is not locked, it's replaced with the address of the sender. The choose of the random neighbor and the replacement operation are processed together thread synchronized. So no other concurrent operation is able to use the same address of the routing table at the same time. Then the peer replies with a message of the type

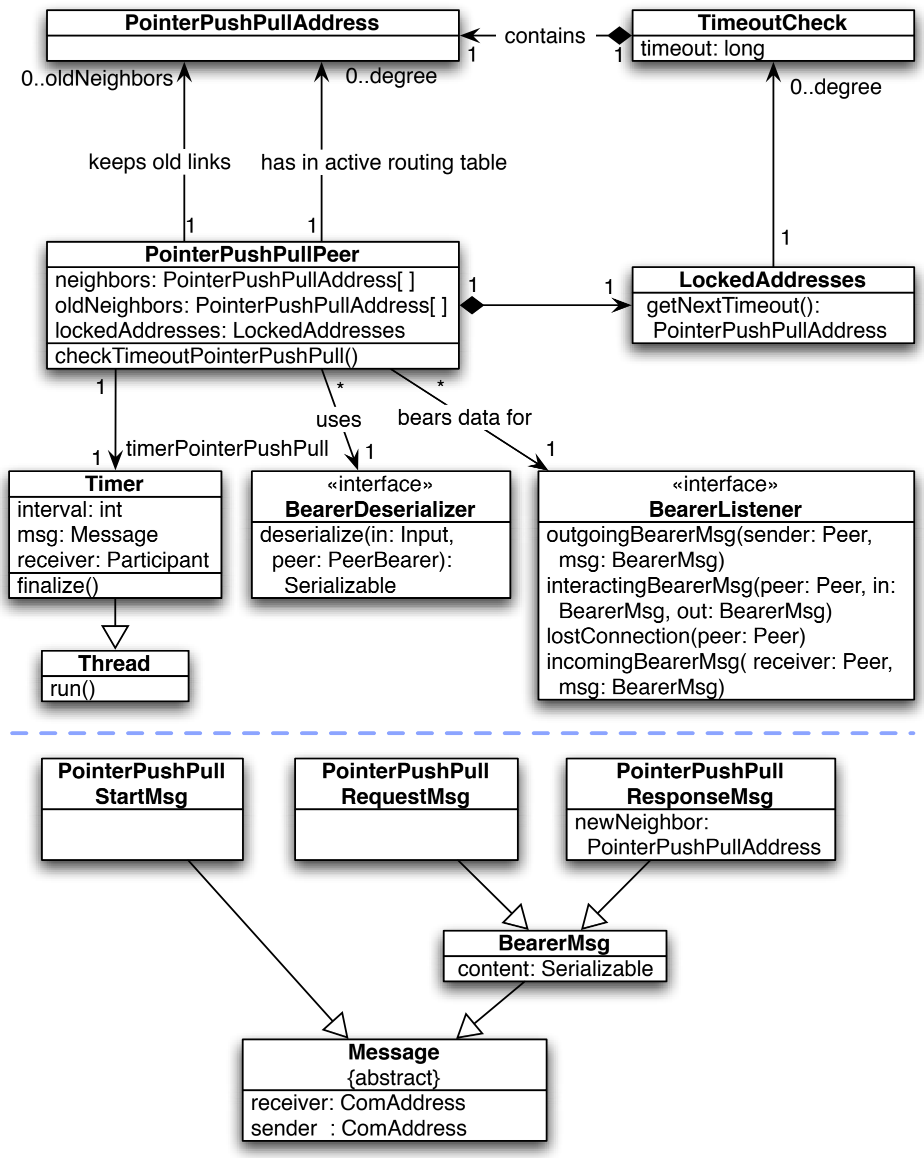

PointerPushPullAddress
contains
TimeoutCheck
timeout: long
0..oldNeighbors
0..degree
keeps old links
has in active routing table
PointerPushPullPeer
neighbors: PointerPushPullAddress[ ]
oldNeighbors: PointerPushPullAddress[ ]
lockedAddresses: LockedAddresses
checkTimeoutPointerPushPull()
LockedAddresses
getNextTimeout(): PointerPushPullAddress
uses
bears data for
timerPointerPushPull
Timer
interval: int
msg: Message
receiver: Participant
finalize()
«interface» BearerDeserializer
deserialize(in: Input, peer: PeerBearer): Serializable
«interface» BearerListener
outgoingBearerMsg(sender: Peer, msg: BearerMsg)
interactingBearerMsg(peer: Peer, in: BearerMsg, out: BearerMsg)
lostConnection(peer: Peer)
incomingBearerMsg(receiver: Peer, msg: BearerMsg)
Thread
run()
PointerPushPull StartMsg
PointerPushPull RequestMsg
PointerPushPull ResponseMsg
newNeighbor: PointerPushPullAddress
BearerMsg
content: Serializable
Message
{abstract}
receiver: ComAddress
sender : ComAddress

Figure 3.10.: Class diagram of the structure for the Pointer-Push&Pull operation.

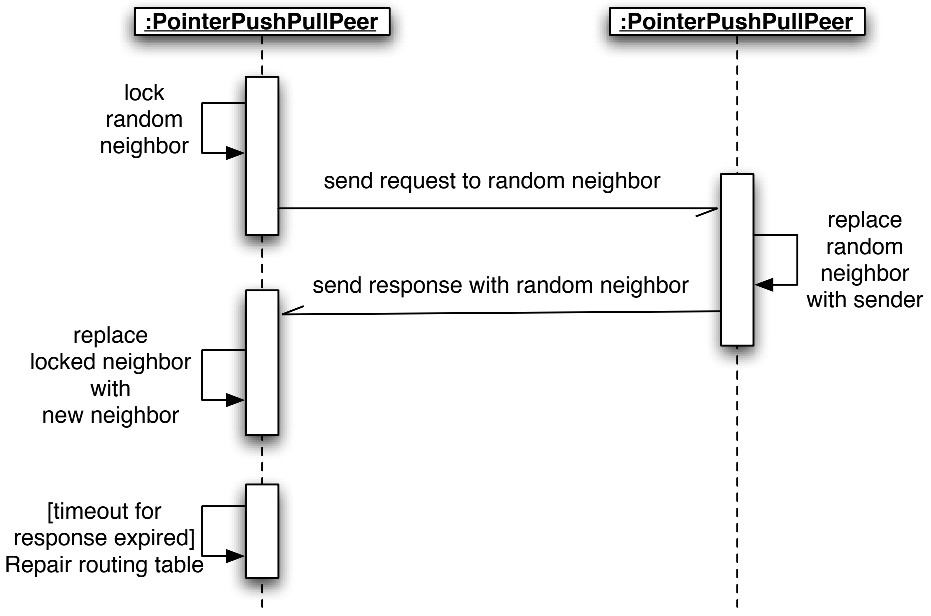

:PointerPushPullPeer
:PointerPushPullPeer
lock random neighbor
send request to random neighbor
replace random neighbor with sender
send response with random neighbor
replace locked neighbor with new neighbor
[timeout for response expired] Repair routing table

Figure 3.11.: Sequence diagram for the Pointer-Push&Pull operation.

PointerPushPullResponseMsg. This response contains the random address which has been replaced in neighbors. If the routing table was empty then the response is sent back without an additional address and the operation is canceled, too.

If a peer receives a response message with the address of a new neighbor, it removes the locked address of the sender of the response from its routing table LockedAddresses. The transmitted new neighbor is added to the active routing table neighbors. The Pointer-Push&Pull operation is successfully finished. But if the message does not contain an additional address of a new neighbor, the locked address is simply unlocked and moved from the data structure LockedAddresses to neighbors. In this case the peer knows that the other peer is currently in the network but could not execute the Pointer-Push&Pull operation at the moment and the operation is canceled.

If the Pointer-Push&Pull operation has been executed successfully, the address of a peer has been removed from the routing table, which has been been checked just in the moment. So the peer knows that the other peer is currently in the network and the quality of this link is in relation to the connectivity of the network very high. So it is useful to keep this link for a while for the case that it's determined that the routing table is incorrect. This addresses are stored in a stack with the name oldNeighbors. The stack has got a maximum size which can be set in the network properties PointerPushPullProperties. If it's full and a new address should be added, the address, which is added first, is deleted.

The maintenance functionality of this operation consists of two parts. The first part is that the Pointer-Push&Pull operation with its simple protocol of a request and a response message can be seen as a ping, too. For this purpose, the locked addresses in the data structure LockedAddresses are furnished with a timeout. This timeout can be adjusted in the parameters of the network properties. In order to handle the case that the asked peer for a Pointer-Push&Pull operation has left the network and does not reply, a thread is started which checks these timeouts. If a timeout is expired, it's known that the other peer has left the network and the routing table contains a dead link. So the routing table has to be repaired. First all links to the peer, which has left the network, are removed. Then the routing table has to be filled up that it has the regular degree again. Addresses of the data structure oldNeighbors are added to the active routing table neighbors first. If that's not sufficient, a random walk operation is started to get new neighbors. This operation is described in section 3.2.1. The second part of the maintenance functionality takes place implicitly. The second peer in the operation which is called for processing a Pointer-Push&Pull operation, replaces always an address of its routing table with an address of a peer, which has sent him a message just in the moment and it knows that this peer is currently in the network. So old links are replaced bit by bit and possible dead links to peers, which could have left the network, are deleted in this way.

Use of the messages for bearing data between applications

As mentioned in the introduction, a random network is normally used to connect applications in the same context. This applications executes together the same distributed service. So if they want to publicize some information about the service, it's probably not important which other application in same context will receive the information. An example is a Peer-to-Peer Network with a top-down approach. An efficient search structure is used at the top and the single elements of the search structure are connected with a simple network structure like a random network. Thereby it would be possible to exchange information about the search structure in the layer above because all peers ,which are connected in the same random network, provide the same part of the search structure together.

In the Pointer-Push&Pull operation messages are sent to random other peers in the overlay network. This messages can be used for transmitting some other data of the application. The advantage is that the information of two operations with the same receiver can be combined in one network packet and that reduces the network traffic.

For this purpose, there exists a bidirectional connection between a PointerPushPullPeer and the application, which uses that peer. The application is able to set a so called

BearerListener in the PointerPushPullPeer. This listener is informed when messages are sent or received in the Pointer-Push&Pull operation. The application can add additional data to these messages or get the additional data from the received messages. The Pointer-Push&Pull operation has an interacting protocol whereby a request is sent and a response is returned. So it would be additionally useful to allow the applications an interaction, too. As result there are three cases to differentiate.

1. The first one is the request message, which starts a new Pointer-Push&Pull operation. There the procedure outgoingBearerMsg() is called in the BearerListener where the listener is able to add some data to this single outgoing message.
2. If a request message for the operation is received, the peer will send a reply back to the sender of the request. In this case the method interactingBearerMsg() is called in the BearerListener. The listener is able to get the data of the received request message and is able to add some data to the corresponding response message.
3. If a response message is received by a peer, the method incomingBearerMsg() is called. The application is able to get the additional data of this message which could be the response of the interaction.

So it's possible to propagate data to arbitrary other applications and it's also possible to make an interaction between two applications with this technique.

The data, which can be added by the BearerDeserializer, has to implement the interface Serializable. This interface describes simply that an object is able to write its content to an output stream. Vice versa, the application has to describe how to deserialize the additional data in the byte array of a message again. Therefore, the application has to set a BearerDeserializer-object in the PointerPushPullPeer-object. This instance is able deserialize the additional data from an input stream and returns the additional data as object. This object is set in the deserialized message object. The BearerListener gets this message with the additional data.

Network packets

Figure 3.12 shows the content of the network packets for the two messages in the Pointer-Push&Pull operation. The complete network packet consists of the IP header, the UDP header, and the content. In the request message the sender address is composed of the IP in the IP header, the port in the UDP header, the local address in the local address header, and the extensions ID and ApplicationInfo. The elements ID, ApplicationInfo, and the content for the BearerMsg have variable length. This messages are also shown as classes PointerPushPullRequestMsg and PointerPushPullResponseMsg in figure 3.10.

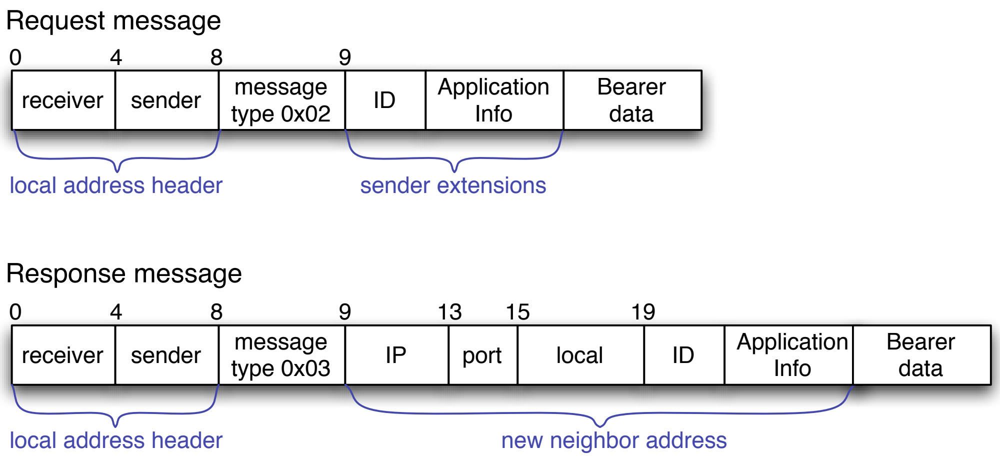


Figure 3.12.: Payload of the network packets in the Pointer-Push&Pull operation.

3.2.3. Common Peer-to-Peer network operations

In this section the main operations of a PointerPushPullPeer are described.

A peer is connected to a random network in the first two operations create and join. Therefor it registers itself in the linked Communication that it is able to send and to receive messages of other peers. The process of the recurrent call of the Pointer-Push&Pull operation is initiated, too. A peer is disconnected from a network in the third operation leave and it de-registers itself in the linked Communication that it will not receive messages anymore. The the execution of the Pointer-Push&Pull-operation is stopped.

Create

The PointerPushPullPeer creates a new random overlay network. The parameters with the type PointerPushPullProperties are used which are set in the peer. Theoretical the first peer of a random network would start with links to itself that it has the regular degree. But this conflicts with the problem that it doesn't know its own address. This problem has been described in section 3.1.2 about restrictions to NAT. As result the first peer starts with an empty routing table and if the second peer joins and contacts the first peer, the first peer will fill up its routing table with links to the second peer. So a peer has only the regular degree if it's not alone in the network.

Join

The PointerPushPullPeer wants to join a still existing random network and has the ComAddress of a peer of that network as parameter. On the one hand the peer requires the properties of the network that it has the same behavior like the other peers. On the other hand it needs addresses of other random peers in the network that it's connected to the network and has the regular degree which is given in the properties. The random walk operation is used to get this information. This operation is described in section 3.2.1. In this operation the peer gets addresses of of other random peers in the random network. The first message of this operation is marked for the join-operation that it will return additionally the parameters of the network. This first message is sent to the address of the known peer, which is given as parameter in the join-operation. The network properties, which are currently set in the peer, are transcribed with the new properties of network, which the peer has joined to.

Leave

The peer is connected to a random network. In order to leave the network again the routing table of the peer is simply cleared. Other peers of the network are not informed that the peer has left it. There will be some dead links to this peer, but they will be fixed with the maintenance functionality of the Pointer-Push&Pull operation, which is called by the other peers of the network.

Lookup

A random network is not suitable for a huge distributed search structure like other Peer-to-Peer networks, because the structure of the overlay network is random and there are no defined ways how to find efficiently a peer which is responsible for a certain element. But it's possible to separate the peers of the random network in a small amount of different ID's. Thereby several peers have the same ID and the assignment of different ID's to the peers is uniformly distributed. As shown in the class diagram in figure 3.8 the ID can be set in the PointerPushPullPeer by the application. The type of the ID is arbitrary and can be defined by the application, too. As price for the generality the application has to define additionally how to deserialize the ID again.

If the regular degree is high enough in ratio to the size of the ID space, then the probability, that a direct neighbor has got the searched ID, is not very small. If that's not the case, a random walk can be passed through until a peer with the searched ID has been found.

The probability that at least one neighbor of a peer has the searched ID is

$$1 - \left(1 - \tfrac{1}{k}\right)^d = 1 - \left[\left(1 - \tfrac{1}{k}\right)^k\right]^{\frac{d}{k}} \approx 1 - e^{-\frac{d}{k}}$$

where d is the regular out-degree and k is the size of the evenly distributed ID space. So the expected length of a random walk for the search is

$$\frac{1}{1-e^{\frac{d}{k}}}$$

So the expected runtime of the search is constant and depends only on the degree of the random network and the size of the ID space.

The lookup operation is a synchronous function which returns the address with the type PointerPushPullAddress of the peer with the searched ID. In this operation it cannot be determined, if actually no peer with the searched ID exists, because the whole random network has to be checked for this conclusion. As result the length of the random walk has to be limited. This value can be set in the network properties PointerPushPullProperties. And an exception with the type PeerNotFoundException is thrown in the lookup operation if no peer with the searched ID could be found.

3.3. Broadcast service

This implementation of a broadcast service is designed to execute broadcasts of rumors in an arbitrary overlay network. The structure of the service is represented in the class diagram in figure 3.13. A rumor represents any data, which an application wants to broadcast is the network. This data has to implement the interface Rumor. A rumor can be broadcasted in the overlay network by calling the function addRumor(Rumor) in a broadcast service with the class type BroadcastService. An application can register itself as listener with the type BroadcastListener in a broadcast service. The broadcast service passes all new rumors to this listener.

In order to embed the broadcast service in an overlay network each peer of the network has to bind an instance of the broadcast service. This association of a peer and a broadcast service fulfills two purposes:

- The peer supplies the broadcast service with random addresses of its routing table. The broadcast service sends the rumors to that random links in order to broadcast the rumors.
- The broadcast service uses the Communication of the peer in order to spread rumors to the broadcast services of other peers. This contains on the one hand that the peer has to implement the specific interface PeerBroadcast. In this interface the peer provides the broadcast service a method to send messages with the network connection of the peer. On the other hand if a peer receives a message, which is actually intended for the broadcast service, the peer must pass the message to the broadcast service.

The messages, which are sent by a broadcast service, are addressed to another random peer instead of the broadcast service of that peer. So a peer must be able to receive messages of a broadcast service and to pass them to the own broadcast service. The types of the messages of the broadcast service are mapped to a byte in the serialization. So this implementation is designed for peers which use a MessageCreator.ByteMap for the deserialization of messages. This adapts to the implementation of the Pointer-Push&Pull Peer which is described in section 3.2. In this scheme the peer registers each different message type with an unique key in a MessageCreator.ByteMap for the deserialization process. The peer has additionally to register the message types of the broadcast service that it's able to receive them. The broadcast service uses three different message types. The peer has to register this message types with a self defined key. This 3 defined keys have to be set in the broadcast service that the broadcast service uses this keys in the sent messages. The keys are shown as attributes in the class BroadcastService in the class diagram in 3.13.

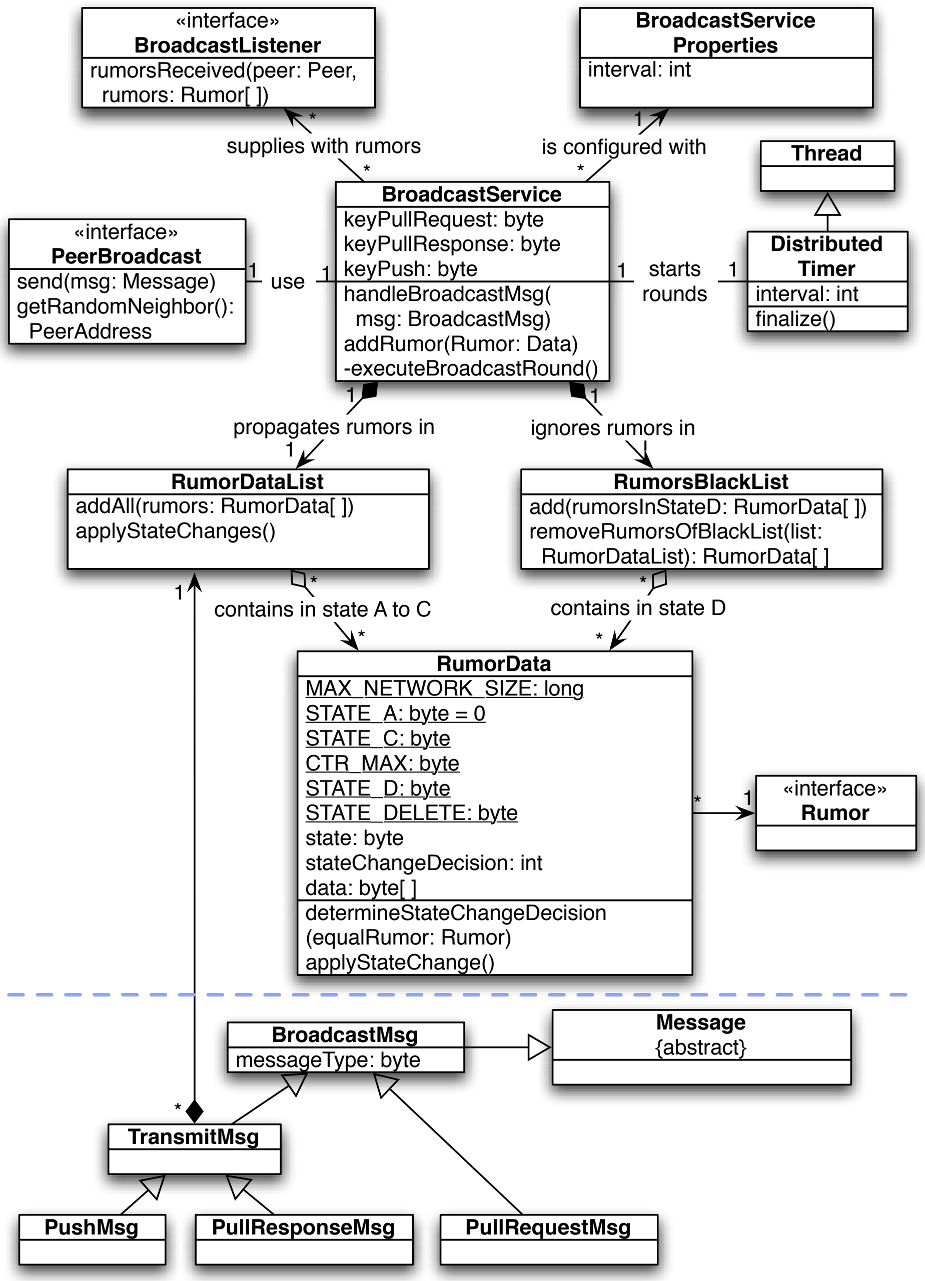


Figure 3.13.: Class diagram of the broadcast service.

As presented in section 2.2 the push&pull-scheme is used to spread rumors. In this scheme each player, here broadcast service, chooses a random link and both players of this link exchange all its rumors.

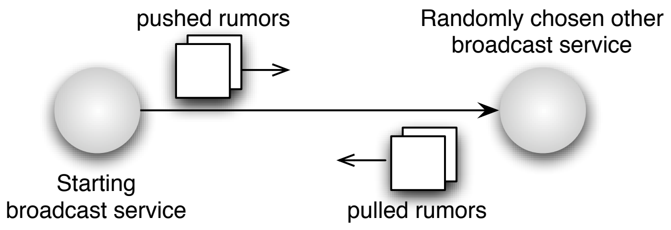

pushed rumors
Randomly chosen other broadcast service
Starting broadcast service
pulled rumors

Figure 3.14.: Exchange of rumors in the broadcast service

3 different messages types are needed for this scheme:

- Push message
- Pull request message
- Pull reply message

Thereby the messages for the push and the pull response contain several rumors. The pull request message has no additional content. If a broadcast service initiates a broadcast and has rumors for spreading, it sends them with a push message else it sends only a pull request message. The other randomly chosen broadcast service, which receives one of those messages, replies with a pull reply message if it has own rumors to propagate. In this way both broadcast services have exchanged all its rumors.

The process of spreading a single rumor is administrated in a RumorData-object. This object contains the rumor as an object, the rumor in the serialized form as a byte array, and the state of the median-counter algorithm. A rumor is sent several times in the process. So it makes sense to serialize it one time and use this byte array for all transmissions. Additionally the byte data of a rumor is used here for comparing two rumors. The byte data can be seen as one big number. Two rumors are equal if both have the same value. In the service several different rumors can be spread at the same time. So if a rumor has been received, it has to be checked first if that rumor has already been received before. And if the same rumor has been found then the process for stopping the spread of the rumor can be executed.

The administration of all rumors takes place in two data structures. Both are bound to the BroadcastService instance. The first one is a RumorDataList which contains all rumors as RumorData. All this rumors are currently spread by the broadcast service. The second data structure has the type RumorsBlackList. This data structure administrates all rumors which are not any longer broadcasted. So if some rumors are received, all rumors are filtered out first which are contained in that data structure.

The broadcast of rumors is round based. In order to distribute the network traffic evenly, a timer of the type DistributedTimer is used. This thread object initiates the broadcast process in an interval and starts the rounds randomly distributed in the time-frame of the interval. In each round the method BroadcastService.executeBroadcastRound() is executed which consists of the following parts:

1. It's first checked if there are rumors which are not any longer broadcasted. This mechanism is described in the next paragraph more precisely. This rumors are moved from the active list in the class RumorDataList to the list with stopped rumors RumorsBlackList.
2. Then it's checked if some rumors which are no longer broadcasted can be deleted. This can be done first when it's secure that the rumor will not received any more. If a deleted rumor would be received again, the broadcast service would assume that it is a new rumor and would start spreading the rumor once again.
3. Then the push&pull operation is executed. The broadcast service gets the address of another random peer in the network from the associated peer. If the service has own rumors which it can spread, it sends a push message with all its rumors to that address. Else it sends a simple pull request message. The service will receive a pull response asynchronous after the execution of this function. This reply contains all rumors of the randomly chosen other broadcast service. If the other service has no rumors to spread, it does not reply.

The median-counter algorithm is used for stopping the broadcast of a rumor. This algorithm is executed for each individual rumor. First of all the maximum network size has to be set in the constant RumorData.MAX_NETWRORK_SIZE in the implementation. The transition between the single states of the algorithm are depended on this value. The transition of the states of the algorithm is proportional to $\log\log(n)$ with the network size n. So it's a parameter which has only a few possible values in the pratical usage. Here this constant is set to 2^{2^5} which can be used for all practicable network sizes. With the double logarithm a smaller value could only be used in very small network with the next order of magnitude of $2^{2^4} = 65365$, $2^{2^3} = 256$, and so on. The constants of the other states in the class RumorData are adjusted to that network size.

If a rumor is received the first time, it's added to the data structure RumorDataList in state A. Else if the same rumor still exists and is currently propagated, the state of both same rumors is compared in order to get a decision if the state of the rumor has to be changed. This is done in the function determineStateChangeDecision(equalRumor: RumorData) in the class RumorData. The decision, if the state should be changed, is kept in the attribute stateChangeDecision. If the received rumor is in state C, this value is set to the maximum value. Else if both rumors are in state B-m, then this value is incremented if the received rumor has a greater or equal B-m' and it's decremented if the B-m' of the other rumor is less. At the beginning of each round the transition of the state is performed. If a rumor is in state C, the state attribute serve additionally for counting the rounds in state C and the state is incremented. If a rumor is in state B-m, it's changed to state C if the decision value is the maximum value and it is changed to state B-(m+1) if the decision value is greater than 0. After the state transition the decision value is reseted to 0 again for the next round. After all state transitions all rumors which are now in state D are moved from the data structure with all current propagated rumors to the data structure RumorsBlackList with all ignored rumors. In each round it is checked if rumors of the black list can be deleted. For that the state attribute is used for counting the rounds in state D, too. In each round this value is incremented in state D and when the maximum number of rounds for storing are expired, the rumor is deleted finally.

4. Bibliography

[1] R. Karp, S. Shenker, C. Schindelhauer, and B. Vöcking. Randomized rumor spreading. In *FOCS'00: 41st Symposium on Foundation on Computer Science*, page 10, 2000.

[2] P. Mahlmann and C. Schindelhauer. Distributed random digraph transformations for peer-to-peer networks. In *SPAA'06: Proceedings of the eighteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, page 10, 2006.

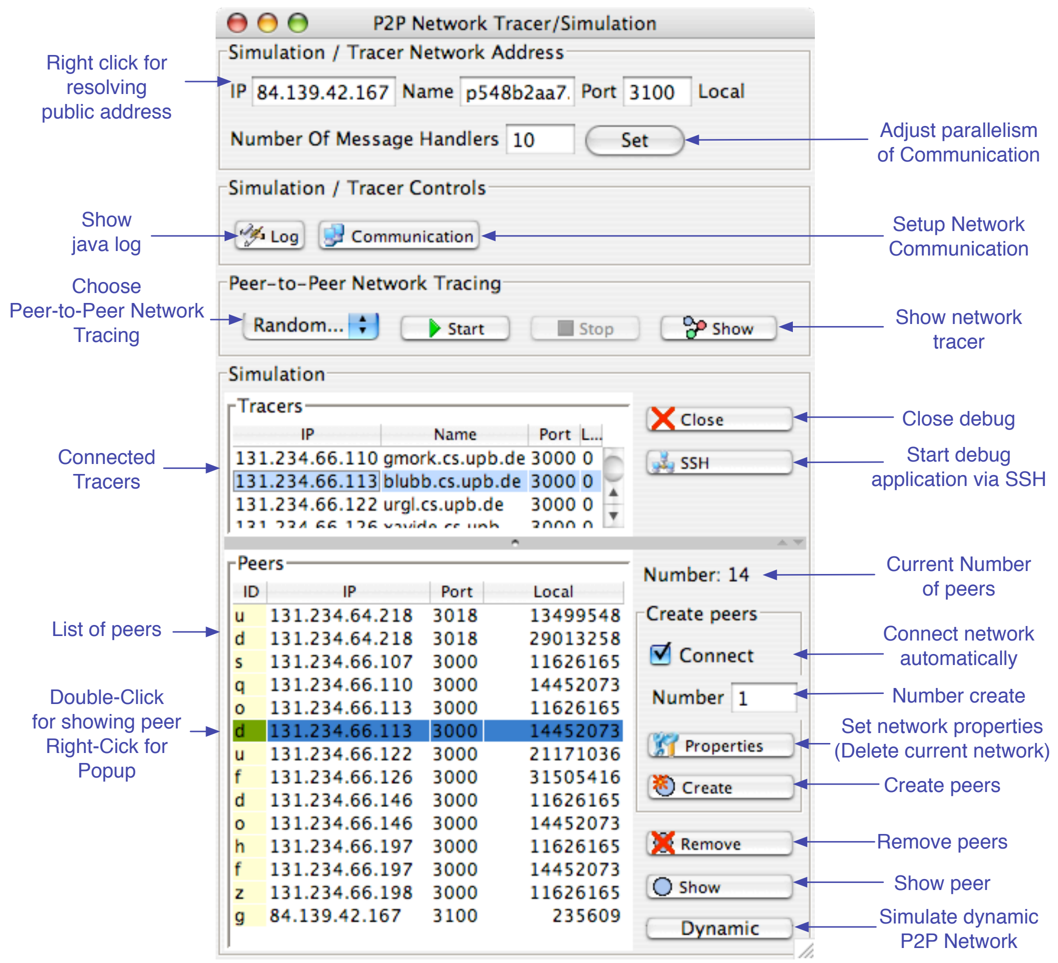
Appendix A.

Test and Diagnostics

In this chapter the test and diagnostics of the implemented Peer-to-Peer Network is shown. The diagnostics contains

- Simulation of a Peer-to-Peer Network
- Trace of a Peer-to-Peer Network

The simulation establishes a dynamic distributed overlay network for test and diagnostics. The implemented tracer for a Peer-to-Peer Network analyzes the state and behavior of this network. The main frame shown in figure A.1 gives overview of all functionalities for the simulation and trace.



P2P Network Tracer/Simulation

Simulation / Tracer Network Address

IP 84.139.42.167 Name p548b2aa7. Port 3100 Local

Number Of Message Handlers 10

Peer-to-Peer Network Tracing: Random...

Tracers

IP	Name	Port	L...
131.234.66.110	gmork.cs.upb.de	3000	0
131.234.66.113	blubb.cs.upb.de	3000	0
131.234.66.122	urgl.cs.upb.de	3000	0

Peers

Number: 14

ID	IP	Port	Local
u	131.234.64.218	3018	13499548
d	131.234.64.218	3018	29013258
s	131.234.66.107	3000	11626165
q	131.234.66.110	3000	14452073
o	131.234.66.113	3000	11626165
d	131.234.66.113	3000	14452073
u	131.234.66.122	3000	21171036
f	131.234.66.126	3000	31505416
d	131.234.66.146	3000	11626165
o	131.234.66.146	3000	14452073
h	131.234.66.197	3000	11626165
f	131.234.66.197	3000	14452073
z	131.234.66.198	3000	11626165
g	84.139.42.167	3100	235609

Create peers: [✓ Connect] Number 1

Right click for resolving public address
Set
Adjust parallelism of Communication
Simulation / Tracer Controls
Show java log
Log
Communication
Setup Network Communication
Choose Peer-to-Peer Network Tracing
Start
Stop
Show
Show network tracer
Simulation
Close
Close debug
SSH
Start debug application via SSH
Connected Tracers
Current Number of peers
List of peers
Connect network automatically
Number create
Double-Click for showing peer Right-Cick for Popup
Properties
Set network properties (Delete current network)
Create
Create peers
Remove
Remove peers
Show
Show peer
Dynamic
Simulate dynamic P2P Network

Figure A.1.: Main frame of the Debug/Simulation of a Peer-to-Peer Network

A.1. Distributed Trace

A framework for tracing a complete distributed Peer-to-Peer Network is implemented. This framework can be extended with tracing elements for the individual network types. Here the tracing extensions for random overlay networks are shown.

A tracer observes peers in order to show the state and the behavior of the peers. In order to analyze a distributed network it is necessary to collect the state of the whole network in one application. For this purpose a structure of several tracers is instantiated. One of them is a special main tracer which has the information of the complete Peer-to-Peer Network. The other simple remote tracers have only the information of the local running peers and provide that information the connected main tracer. This structure is shown in figure A.2.

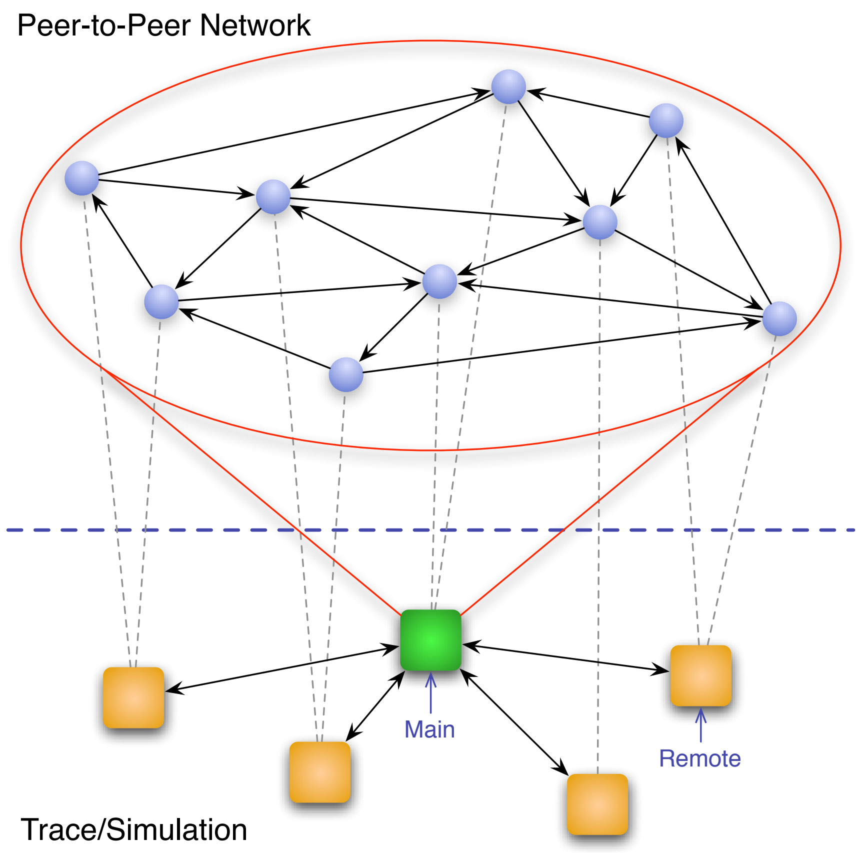


Figure A.2.: Structure of a distributed tracing for a Peer-to-Peer Network

So the network of tracers is centralized with one main tracer and several remote tracers. In order to establish this structure a main tracer has to be started first. Then several remote tracers can be started which get the address of the main tracer as parameter. Each remote tracer sends a message to the main tracer with its own address for a bidirectional connection of the centralized tracer network.

A tracer uses a Communication in order to exchange information with other tracers. A Communication is a network communication model which allows several participants to use a network connection shared. In this model each participant has an extended address which consists of the IP address, the port, and an additional local address within the Communication. A tracer has a fixed local address. So a tracer can be addressed only with the IP address and the port. The Commmunication of a tracer can be modified and extended in the dialog shown in figure A.3. Press the button Communication in the main frame (A.1) to open this dialog.

Normally a factory is used to instantiate a main or remote tracer. This factory creates the type of the debug dependent on a string parameter which can be added to the application parameters. An example for the parameters is shown in the listing below. The first one is the parameter for a main tracer and the second one for a remote tracer. This parameter should not contain any empty spaces.

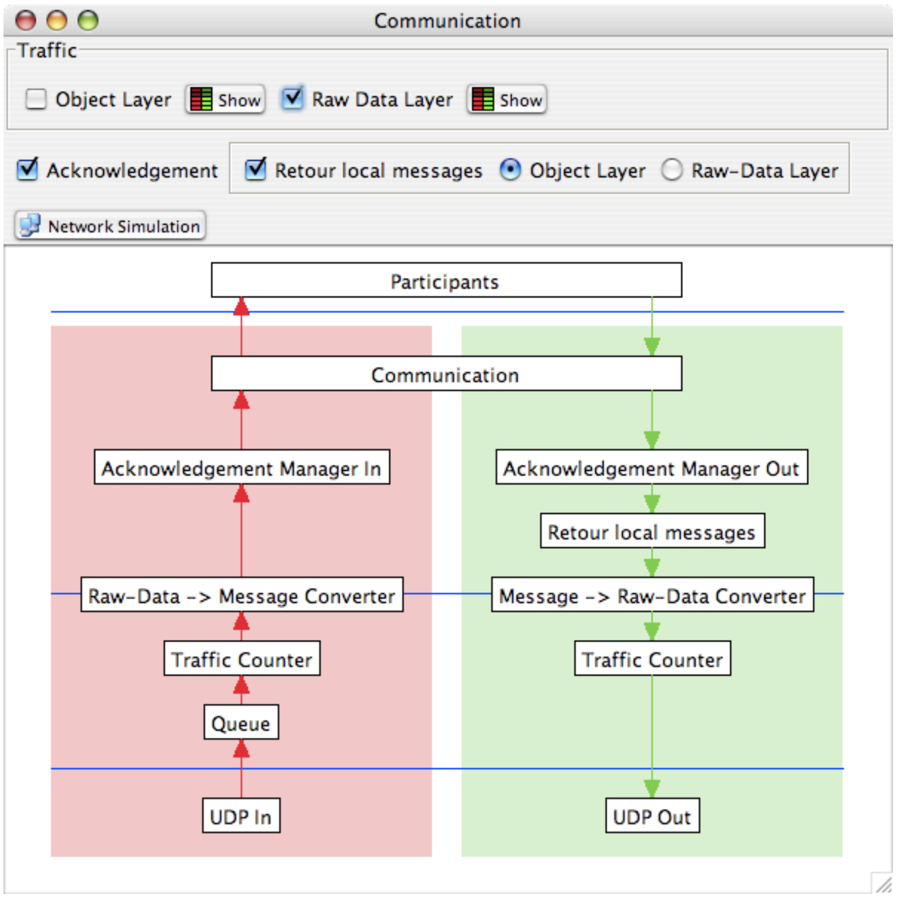
```
-tracer(3000,isSim,Main,create=4)
-tracer(3000,isSim,Remote,127.0.0.1,3000,create=4)
```

The first element is the port of the Communication which is instantiated for the tracer. The second element isSim is optional and is part of the simulation. If it is set, then peers which are removed from the simulation are also deleted and if the simulation frame is closed then the application exits. The third element Main|Remote is the type of the tracer. If the type is Remote (second example), then the following two elements are the address of the main tracer with IP address and port. The remote tracer connects to that main tracer with that address. The last parameter is also part of the simulation. It can be set how many peers should be created initially.

An example for the parameters is given in the scripts for the trace of a random overlay network.

```
Run_RandomNetwork_Sim_Main
Run_RandomNetwork_Sim_Remote
Run_RandomNetwork_Sim_Remote_Show
```

The first one starts a main simulation/tracer for a random network. The other two scripts starts a remote tracer/simulation whereby only the last one is started with a GUI. Thereby the address of the main trace is set to the local host. So call first the



Communication

Traffic

☐ Object Layer ✓ Raw Data Layer

✓ Acknowledgement ✓ Retour local messages [● Object Layer] ○ Raw-Data Layer

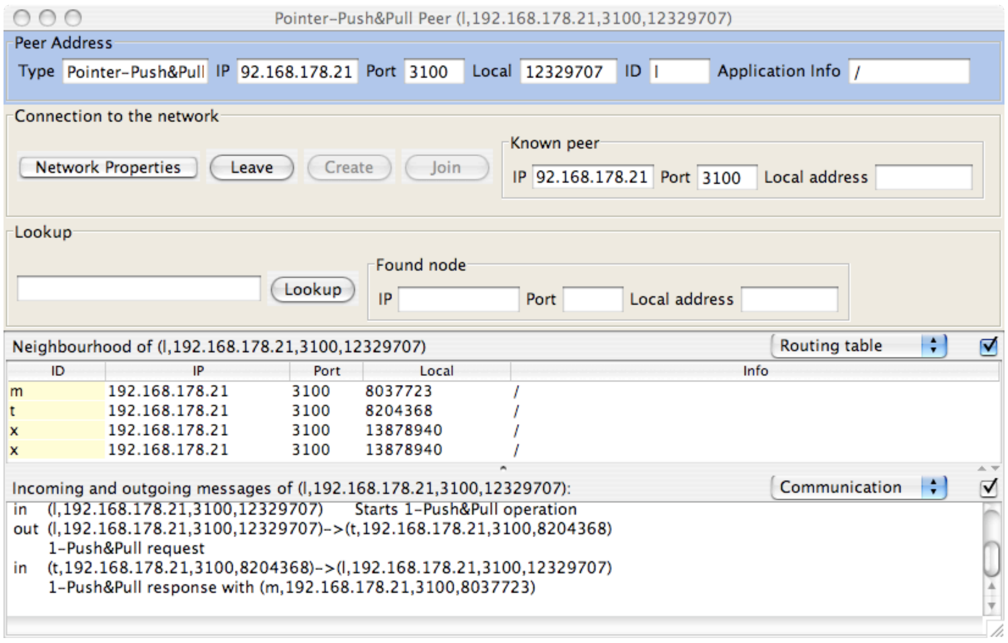
Figure A.3.: Setup of the Communication of the debug.

script for the main tracer and then one of the scripts for a remote tracer on the same computer and both tracers are connected to each other.

A.1.1. Peer Trace

The trace of a single peer can be started in a remote or main trace, because it only depends the information of a single peer. In order to show the trace you have to select a single peer in the list on the bottom of the main frame (A.1) and either you double click on the address of the peer or click on the button Show on the right side of the peer list.

The type of tracing can be defined by the application. Standardly it is set a frame which can contain several tracing elements. Each of that elements shows different information about the peer. Figure A.4 shows this frame for a Pointer-Push&Pull Peer in a random overlay network. In the middle you can see the routing table which is a multi-set. On the bottom the message exchanges of the peer are shown.



Pointer-Push&Pull Peer (l,192.168.178.21,3100,12329707)

Peer Address
Type Pointer-Push&Pull IP 92.168.178.21 Port 3100 Local 12329707 ID l Application Info /

Known peer
IP 92.168.178.21 Port 3100 Local address

Found node
IP Port Local address

Neighbourhood of (l,192.168.178.21,3100,12329707)

ID	IP	Port	Local	Info
m	192.168.178.21	3100	8037723	/
t	192.168.178.21	3100	8204368	/
x	192.168.178.21	3100	13878940	/
x	192.168.178.21	3100	13878940	/

Incoming and outgoing messages of (l,192.168.178.21,3100,12329707):

```
in   (l,192.168.178.21,3100,12329707)      Starts 1-Push&Pull operation
out  (l,192.168.178.21,3100,12329707)->(t,192.168.178.21,3100,8204368)
     1-Push&Pull request
in   (t,192.168.178.21,3100,8204368)->(l,192.168.178.21,3100,12329707)
     1-Push&Pull response with (m,192.168.178.21,3100,8037723)
```

Figure A.4.: Visualization a single Pointer-Push&Pull Peer

The connection between peers and a tracer is established with a listeners or observer concept. Thereby it is possible to set an oberserver in the peer for

- the message exchange of a peer
- changes of the peer state (routing table)

If a peer sends or receives a message or changes its routing table, it informs the observer. Then the observer like the tracer is able to present the current state and the behavior of the peer.

A.1.2. Peer-to-Peer Network Trace

The trace of a whole Peer-to-Peer network can only be processed in a main tracer. A main tracer has the information of the whole network. For this purpose a Virtual Peer is implemented(figure A.5). This structure is based on a proxy pattern. The real peer runs

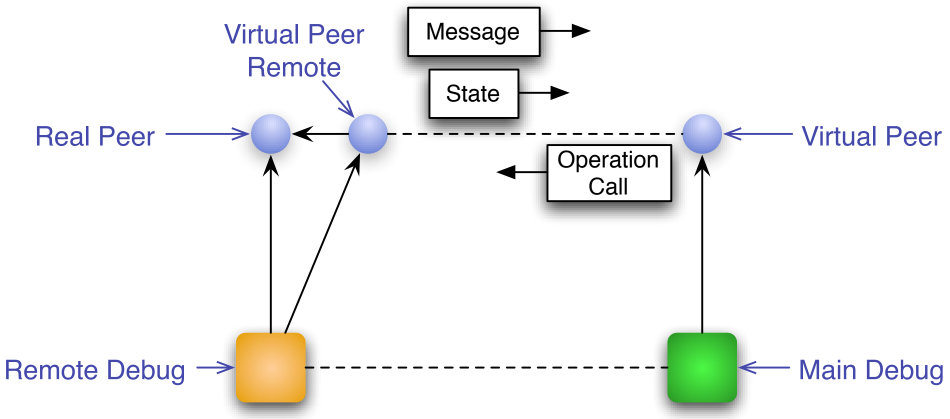


Figure A.5.: Structure of a Virtual Peer

remotely referenced in a remote tracer. The remote tracer creates a Virtual Peer Remote. Then it sends the address of the Virtual Peer Remote to the main tracer. The main tracer creates a Virtual Peer which is connected in the network with the Virtual Peer Remote. The Virtual Peer and the Virtual Peer Remote participates in the Communication of the tracer.

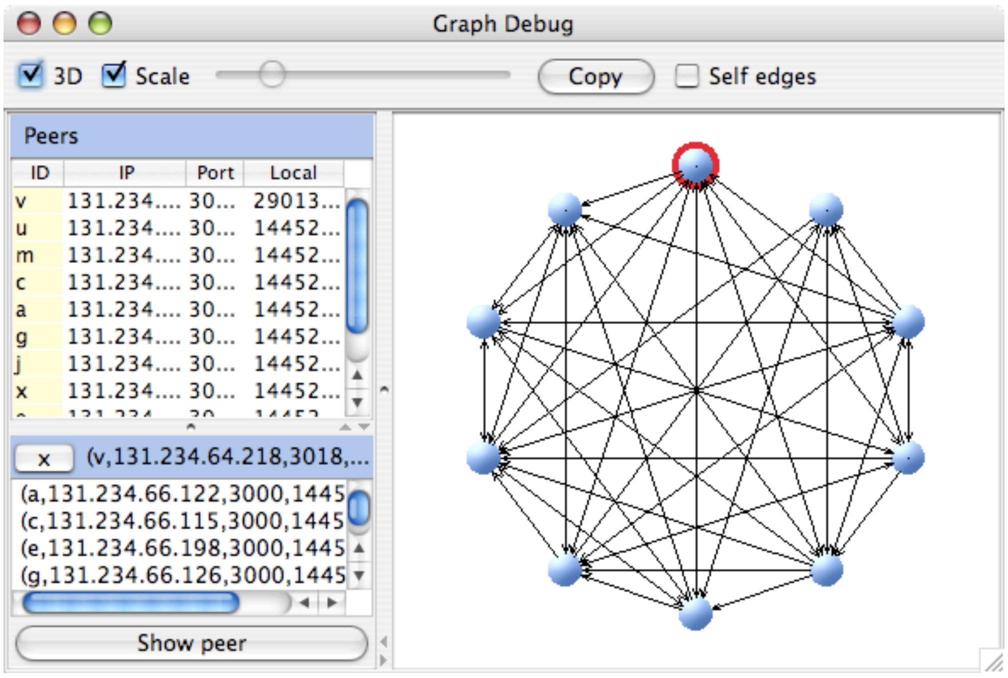
If the main tracer calls an operation in the Virtual Peer then this operation call is forwarded in the network to the Virtual Peer Remote. The Virtual Peer Remote calls the operation in the real peer. The Virtual Peer Remote is additionally able to register itself as observer in the real peer. If the real peer informs the Virtual Peer Remote about a message or state change, then this information is sent to the Virtual Peer. The main tracer is able to set an observer in the Virtual Peer an gets the transmitted changes of the peer in this way.

So the main tracer has got a representative of each peer in the Peer-to-Peer Network:

- If the peer runs in the local application, the real peer is referenced.
- If the real peer runs in another application, a Virtual Peer is referenced

And with the state of each peer in the network it's of course possible to present and analyze the complete Peer-to-Peer Network.

Several tracing elements, which present information of the traced Peer-to-Peer Network, can be added to the main tracer. That elements are shown in the third section Peer-to-Peer Network Tracing of the main frame (A.1). There it is possible to choose one of the tracing elements in a drop down box. The buttons beside the drop down box start, stop or show the selected tracing tool. Figure A.6 shows an example for the visualization of random overlay network. In this frame you can click or double-click on a peer in the graph or in the list on the left to get additional information for a single peer. Another example is presented in figure A.7 where the degree of a random overlay network can be analyzed.



✓ 3D ✓ Scale ☐ Self edges

Peers

ID	IP	Port	Local
v	131.234....	30...	29013...
u	131.234....	30...	14452...
m	131.234....	30...	14452...
c	131.234....	30...	14452...
a	131.234....	30...	14452...
g	131.234....	30...	14452...
j	131.234....	30...	14452...
x	131.234....	30...	14452...

(v,131.234.64.218,3018,...

(a,131.234.66.122,3000,1445
(c,131.234.66.115,3000,1445
(e,131.234.66.198,3000,1445
(g,131.234.66.126,3000,1445

Figure A.6.: Visualization of a random overlay network

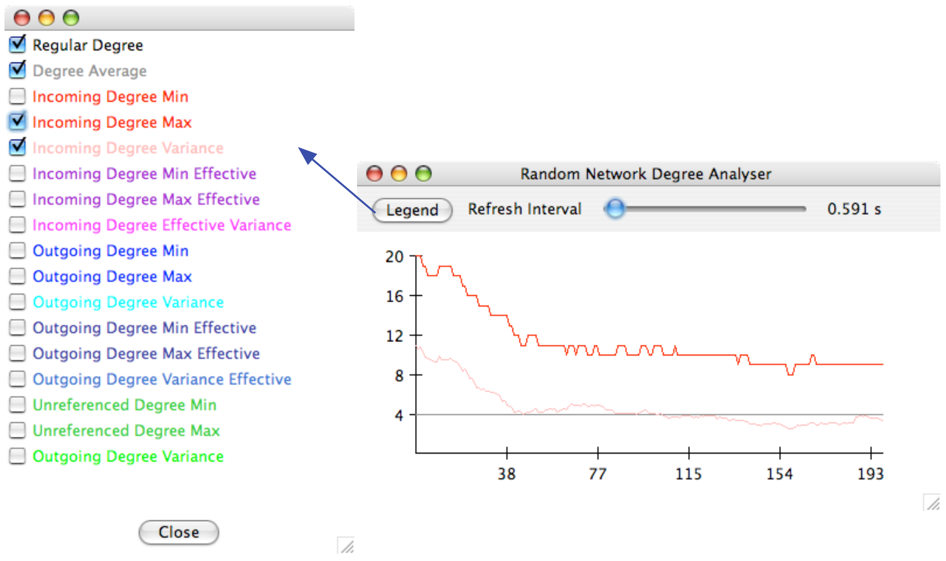

✓ Regular Degree
✓ Degree Average
☐ Incoming Degree Min
✓ Incoming Degree Max
✓ Incoming Degree Variance
☐ Incoming Degree Min Effective
☐ Incoming Degree Max Effective
☐ Incoming Degree Effective Variance
☐ Outgoing Degree Min
☐ Outgoing Degree Max
☐ Outgoing Degree Variance
☐ Outgoing Degree Min Effective
☐ Outgoing Degree Max Effective
☐ Outgoing Degree Variance Effective
☐ Unreferenced Degree Min
☐ Unreferenced Degree Max
☐ Outgoing Degree Variance
Close
Random Network Degree Analyser
Legend
Refresh Interval
0.591 s
20
16
12
8
4
38
77
115
154
193

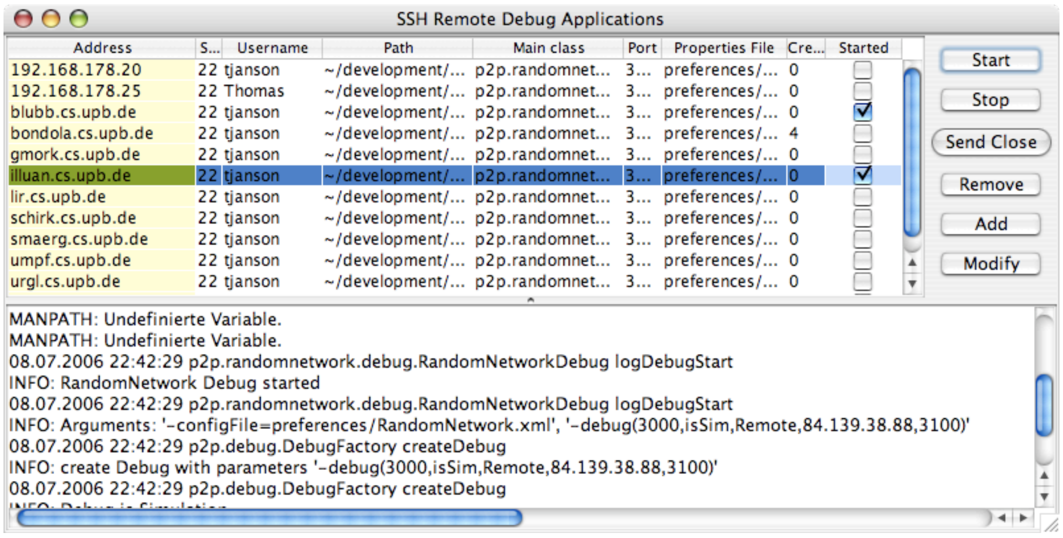
Figure A.7.: Degree analysis of a random overlay network.

A.1.3. SSH

A remote tracer can be executed with or without a visualization. If it's started without a visualization then it is also possible to start that application with a tracer with a SSH remote shell. So the user is able to start several applications on different computers from one single computer and establishes a real Peer-to-Peer Network.

For this purpose, a tool has been created that automates the starting of a java application via a SSH connection on another computer. In this tool the application information for several computers can be setup. The user can select these setups in a list and start them with one mouse click. Figure A.8 shows the GUI of that tool. The tool can also be opened with the button SSH in the tracer main frame A.1).

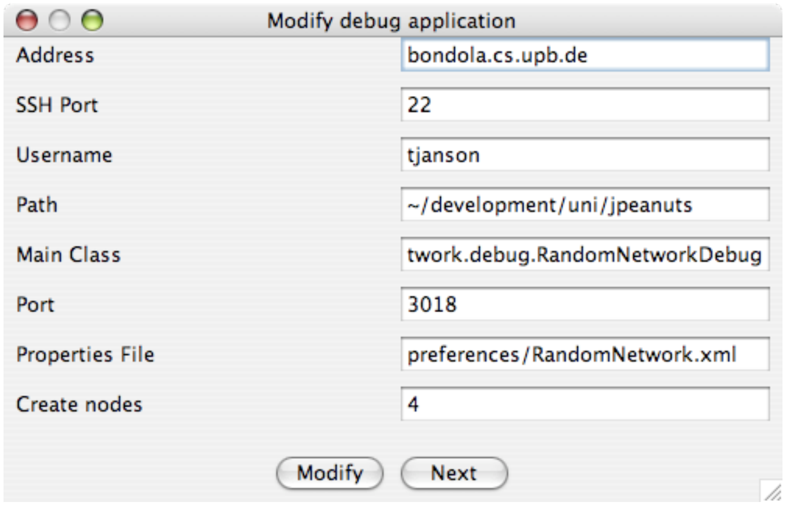
The button Add can be pressed for creating a new setup of a java application via SSH. Then the dialog shown in figure A.9 pops up. The first three parameters Address, SSH Port, and Username are used for establishing the SSH connection. Standardly the port for SSH is 22. But if several computers have the same IP address, then it has to be differentiated between the computers with the port. This case appears when several computers use a shared internet connection with a router. The security check for the Man-In-The-Middle-Attack is disabled in SSH because several computers with the same



Address	S...	Username	Path	Main class	Port	Properties File	Cre...	Started
192.168.178.20	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
192.168.178.25	22	Thomas	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
blubb.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	✓
bondola.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	4	☐
gmork.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
illuan.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	✓
lir.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
schirk.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
smaerg.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
umpf.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐
urgl.cs.upb.de	22	tjanson	~/development/...	p2p.randomnet...	3...	preferences/...	0	☐

```
MANPATH: Undefinierte Variable.
MANPATH: Undefinierte Variable.
08.07.2006 22:42:29 p2p.randomnetwork.debug.RandomNetworkDebug logDebugStart
INFO: RandomNetwork Debug started
08.07.2006 22:42:29 p2p.randomnetwork.debug.RandomNetworkDebug logDebugStart
INFO: Arguments: '-configFile=preferences/RandomNetwork.xml', '-debug(3000,isSim,Remote,84.139.38.88,3100)'
08.07.2006 22:42:29 p2p.debug.DebugFactory createDebug
INFO: create Debug with parameters '-debug(3000,isSim,Remote,84.139.38.88,3100)'
08.07.2006 22:42:29 p2p.debug.DebugFactory createDebug
```

Figure A.8.: Frame for the automatically start of the debug via SSH



Address	bondola.cs.upb.de
SSH Port	22
Username	tjanson
Path	~/development/uni/jpeanuts
Main Class	twork.debug.RandomNetworkDebug
Port	3018
Properties File	preferences/RandomNetwork.xml
Create nodes	4

Figure A.9.: Setup for a trace application via SSH

IP address but different ports will have different fingerprints and the connection to more than one computer with the same IP address would fail.

All further parameters of the input mask are related to the application that should be started on the server via SSH. As precondition the java application has to be available on the server in the compiled form. Then the parameter Path defines the path of the application on the server. The compiled class files of the application should be in the subdirectory bin in this path. The parameter Main Class defines the class with the main function which should be started. In the example of the random overlay networks this class is p2p.randomnetwork.debug.RandomNetworkDebug.

The last three parameters Port, Properties File, and Create Nodes are related to the parameters of the java application. The port is the port of the Communication which is instantiated by the tracer. The properties file contains the properties of the Peer-to-Peer Network. The last parameter Create Nodes defines how many peers should be created initially. This part of the simulation.

It is possible to modify several setups on the same time. For this purpose you can select several setups in the table and click on the button Modify. Then all parameters excepting the network address are overwritten.

The buttons Start and Stop starts and stop processes for the execution of debug applications via SSH. Here it is also possible to select several setups at the same time. On the bottom of the frame the output of one of the started SSH processes is shown. You can double click on a setup in order to change to that output. The double click on the same element again stops the output in the frame.

It is always a remote debug started in this context which contains the address of the main debug as parameter. Then the remote debug connects to the main debug so that it can be controlled by the main debug. If the tool is started in the main debug then the address of the main debug is automatically used. If the application is on a computer behind a router, then it is necessary to resolve the public address of the router first. For this purpose you can make a right click on the IP address in the simulation/debug frame and a popup will be shown. In the popup the option Refresh the IP address with a webservice can be chosen to resolve the public address.

Setup SSH without password

The tool for the automatically execution of java applications via SSH is designed without any additional user input. As result it cannot be entered the password for the login to the server. In this section it is described how to establish a SSH connection from a client to a server without a password.

Additionally you have to log in each server at least one time before you can use it in the tool. The first time it will be asked if you accept the fingerprint of the server and this has to be replied with yes. If this is not done, then the error "Host key verification failed" will be prompted in the console output of the SSH tool.

Client
First a fingerprint for the client has to be created which consists of a private key and a public key. The public key has the postfix .pub.

```
ssh-keygen -t rsa - b 1024
```

In the execution of the command the following has to be entered:

- Set the the storage location (for example .ssh/ida_rsa)
- Set an empty password

Then the directory /.ssh should contain the public key in the file ida_rsa.pub and the private key in the file ida_rsa. As next step the public key has to be copied to the server.

```
scp ~/.ssh/ida_rsa.pub user@server:~/.ssh/ida_rsa_client.pub
```

Server
First you have to log in the server with password.

```
ssh user@server
```

Then the public key of the client, which has been copied to server, has to be published in the server.

```
cat ~/.ssh/ida_rsa_client.pub >> ~/.ssh/authorized_keys
```

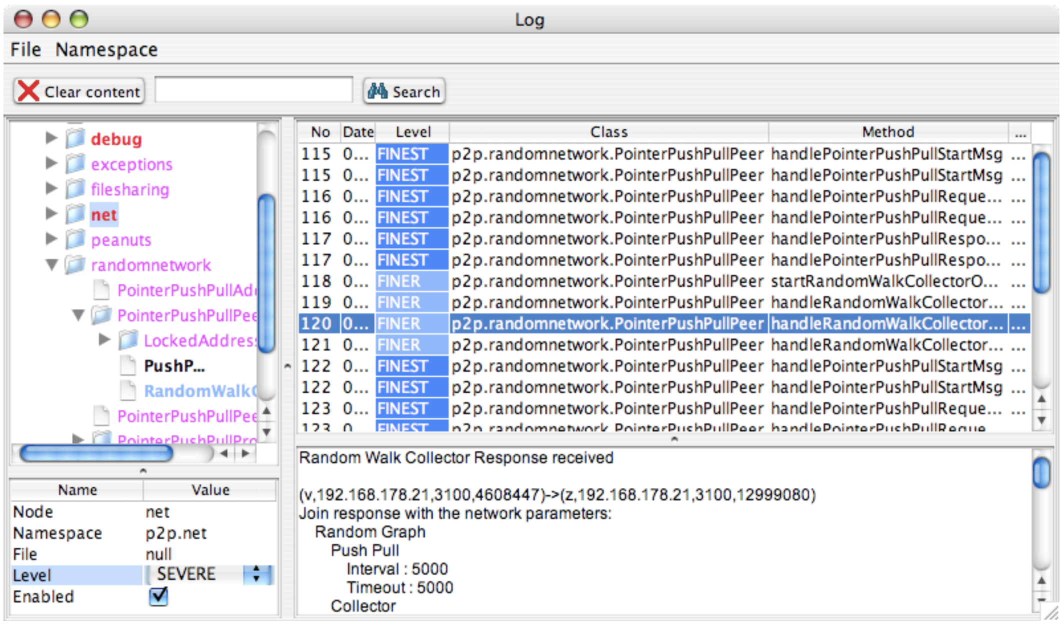
Then you can log out and should be able to login without password again.

A.1.4. Logging

The package java.util.logging is used for logging the course of the program. This package is a port of *log4j* and has the advantage over log4j that it is already included in the standard java packet.

Figure A.10 shows a dialog frame for the representation of the log messages. In the tree view on the left side of the dialog it is also possible to define which elements of the running program should be logged. The course of single operations of the

Pointer-Push&Pull are logged here, too. If you enable the logger with the path p2p.net.randomnetwork.PointerPushPullPeer.PushPull for example, then the execution of the Pointer-Push&Pull operation is logged.



No	Date	Level	Class	Method	...
115	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullStartMsg	...
115	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullStartMsg	...
116	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullReque...	...
116	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullReque...	...
117	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullRespo...	...
117	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullRespo...	...
118	0...	FINER	p2p.randomnetwork.PointerPushPullPeer	startRandomWalkCollectorO...	...
119	0...	FINER	p2p.randomnetwork.PointerPushPullPeer	handleRandomWalkCollector...	...
120	0...	FINER	p2p.randomnetwork.PointerPushPullPeer	handleRandomWalkCollector...	...
121	0...	FINER	p2p.randomnetwork.PointerPushPullPeer	handleRandomWalkCollector...	...
122	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullStartMsg	...
122	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullStartMsg	...
123	0...	FINEST	p2p.randomnetwork.PointerPushPullPeer	handlePointerPushPullReque...	...

Name	Value
Node	net
Namespace	p2p.net
File	null
Level	SEVERE
Enabled	✓

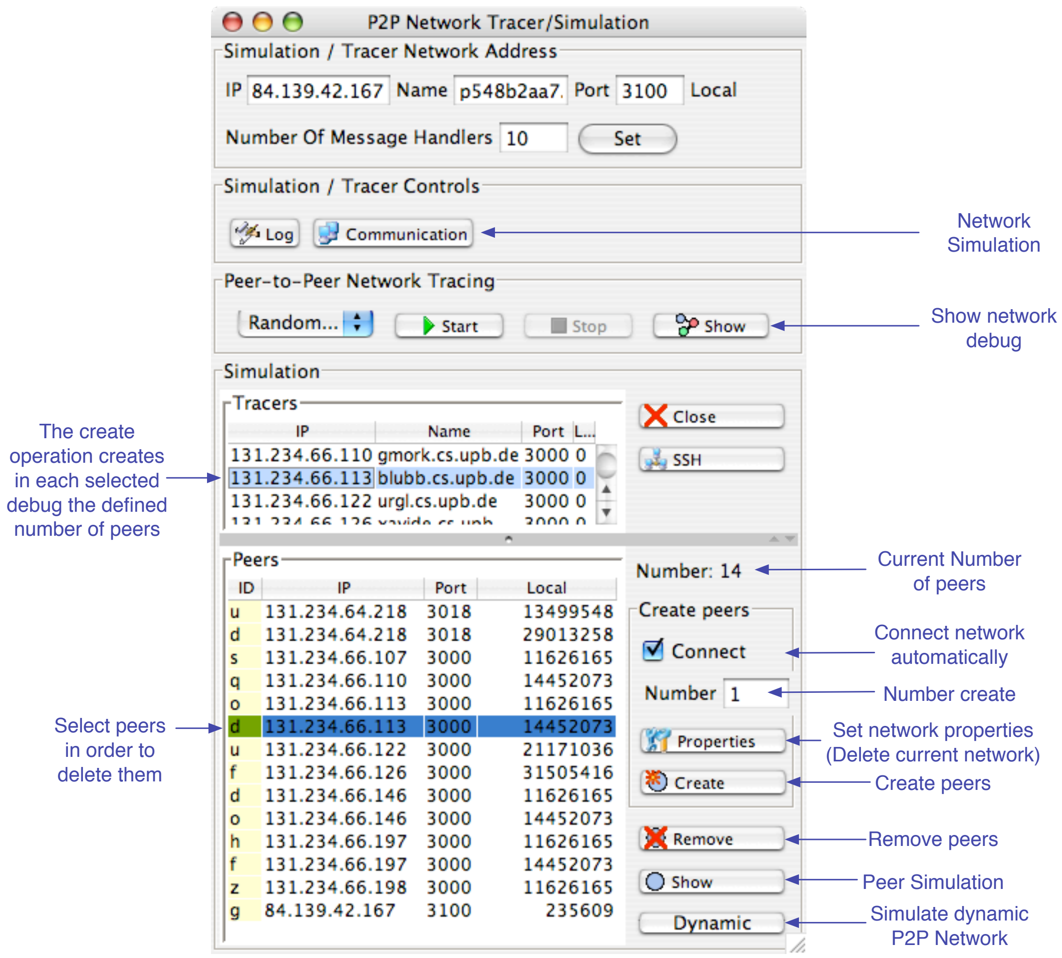
```
Random Walk Collector Response received

(v,192.168.178.21,3100,4608447)->(z,192.168.178.21,3100,12999080)
Join response with the network parameters:
  Random Graph
    Push Pull
      Interval : 5000
      Timeout : 5000
    Collector
```

Figure A.10.: Logging frame

A.2. Simulation

The diagnostics framework is able to create and simulate a dynamic distributed Peer-to-Peer Network. So a real P2P application with real users is not absolutely necessary for the diagnostics. This functionality is contained in the same visualization shown in figure A.11.

A PeerFactory, defined in section 3.1.2, has to be set in the simulation so that the simulation is able to instantiate the peers of the individual Peer-to-Peer Network. If the framework is used also as simulation, then the parameter isSim has to be set in the tracer which is described in A.1. This attribute indicates that the peers, which are referenced in the tracer, are not used by another application and it's possible to delete the peers.


P2P Network Tracer/Simulation
Simulation / Tracer Network Address
Set
Simulation / Tracer Controls
Log
Communication
Network Simulation
Peer-to-Peer Network Tracing
Random...
Start
Stop
Show
Show network debug
Simulation
Tracers
Close
SSH
The create operation creates in each selected debug the defined number of peers
Peers
Number: 14
Current Number of peers
Create peers
Connect
Connect network automatically
Number 1
Number create
Properties
Set network properties (Delete current network)
Create
Create peers
Remove
Remove peers
Show
Peer Simulation
Dynamic
Simulate dynamic P2P Network
Select peers in order to delete them

IP 84.139.42.167 Name p548b2aa7. Port 3100 Local

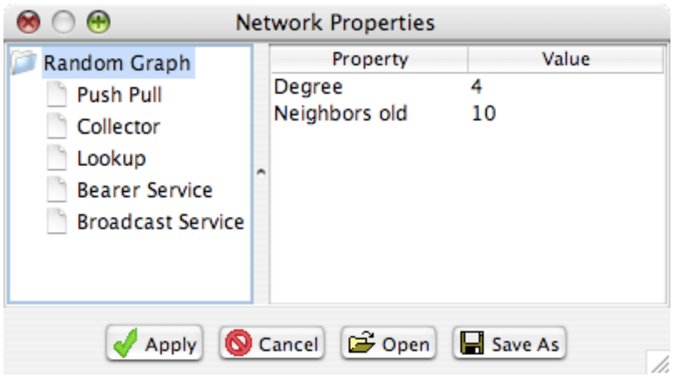
Number Of Message Handlers 10

IP	Name	Port	L...
131.234.66.110	gmork.cs.upb.de	3000	0
131.234.66.113	blubb.cs.upb.de	3000	0
131.234.66.122	urgl.cs.upb.de	3000	0
131.234.66.126	xavide.cs.upb	3000	0

ID	IP	Port	Local
u	131.234.64.218	3018	13499548
d	131.234.64.218	3018	29013258
s	131.234.66.107	3000	11626165
q	131.234.66.110	3000	14452073
o	131.234.66.113	3000	11626165
d	131.234.66.113	3000	14452073
u	131.234.66.122	3000	21171036
f	131.234.66.126	3000	31505416
d	131.234.66.146	3000	11626165
o	131.234.66.146	3000	14452073
h	131.234.66.197	3000	11626165
f	131.234.66.197	3000	14452073
z	131.234.66.198	3000	11626165
g	84.139.42.167	3100	235609

Figure A.11.: Main frame of the Debug/Simulation of a Peer-to-Peer Network

Peers can be created in the lower section of the main frame shown in figure A.11. First you have to define the properties of the network which should be created (Button Properties in the main frame). The whole network has to be deleted before the changes take effect. The properties are only set in new created peers. If the option Connect is enabled, then the peers connect automatically to the



Property	Value
Degree	4
Neighbors old	10

Figure A.12.: Network properties of a Pointer-Push&Pull Peer in a random network

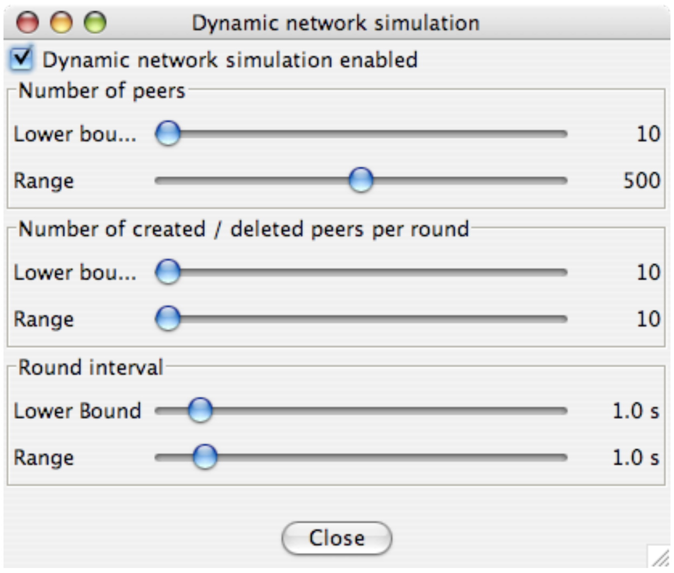
running Peer-to-Peer Network. In this scheme one single connected Peer-to-Peer Network is created. The network can be created distributed on several debugs which are connected to each other (figure A.2). It is also possible to control the remote debugs and peers can be created remotely in a remote debug. For this functionality the defined number of peers is created in each selected tracer. The tracers can be selected in the middle section of the main frame (figure A.11).

In order to delete peers again you can select some peers in the list in the main frame and click on the button Delete.

It is also possible to to start a dynamic network whereby peers are randomly created and deleted. Click on the button Dynamic in the east-south of the main frame and a dialog for that functionality will appear (figure A.13).

Additionally you can call the main operations of a single peer. This operations are defined in section 3.1.2. Double click on the address of a peer in the main frame or select a peer peer and click on the button Show. The dialog shown in figure A.4 will appear. There you are able to call the operations create, join, leave, and lookup.

It's also possible to use a Peer-to-Peer Network trace element (section A.1.2) for the simulation. Figure A.14 shows an example where a broadcast is processed in a random overlay network. There you can selected a peer in the graph or in the



☑ Dynamic network simulation enabled

Number of peers

Lower bou...	10
Range	500

Number of created / deleted peers per round

Lower bou...	10
Range	10

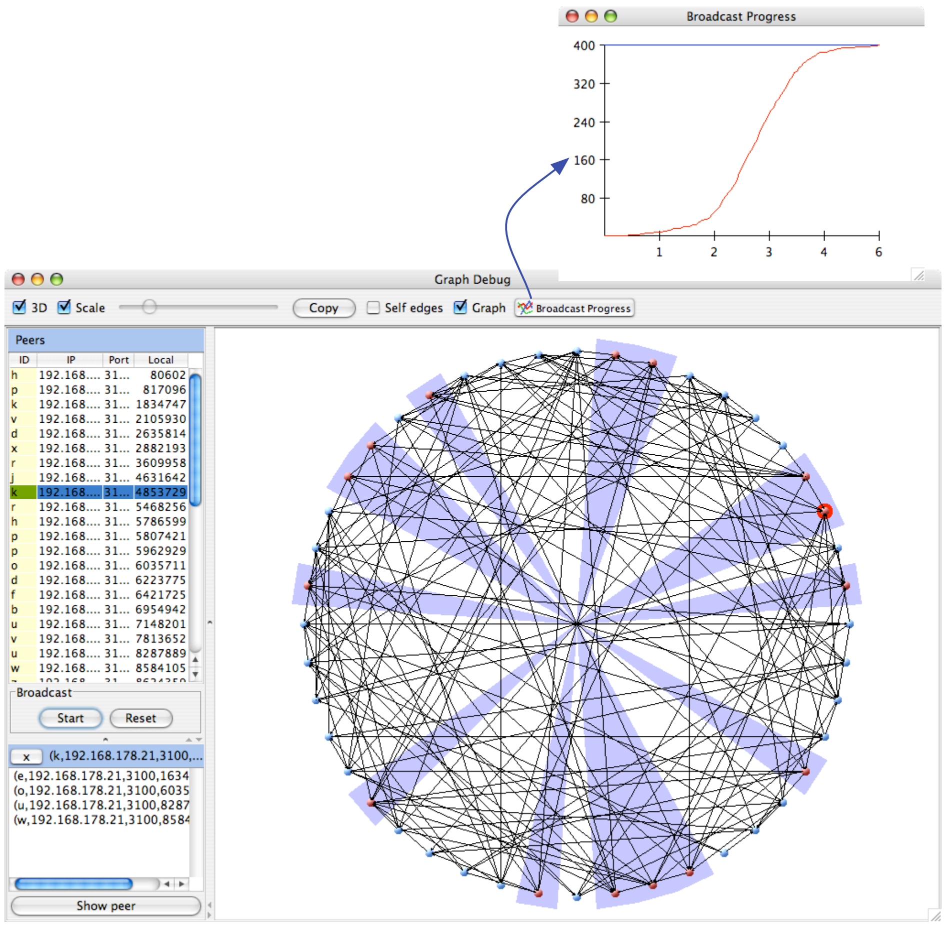
Round interval

Lower Bound	1.0 s
Range	1.0 s

Figure A.13.: Dialog for the simulation of a dynamic network

list on the left and press the button Start in order to start a new broadcast in the selected peer. If the button Reset is pressed, then the current shown debug is ignored and not shown any longer.

The peers, which are created in the simulation, use the Communication of the debug shared. It's possible to manipulate the network connectivity. For this purpose click on Communication in the simulation frame and then the dialog shown in figure A.3 appears. In this dialog you can click on Network Simulation and a dialog shown in figure A.15 is shown. In this dialog you can manipulate the network traffic. This contains that some message disappear randomly or messages are sent with a delay.

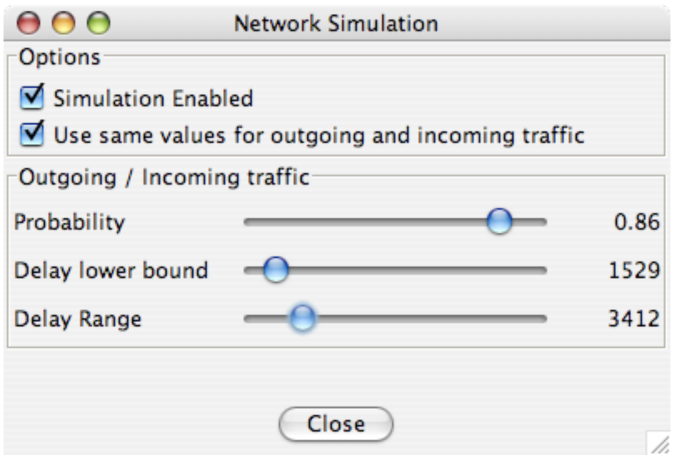


Peers

ID	IP	Port	Local
h	192.168....	31...	80602
p	192.168....	31...	817096
k	192.168....	31...	1834747
v	192.168....	31...	2105930
d	192.168....	31...	2635814
x	192.168....	31...	2882193
r	192.168....	31...	3609958
j	192.168....	31...	4631642
k	192.168....	31...	4853729
r	192.168....	31...	5468256
h	192.168....	31...	5786599
p	192.168....	31...	5807421
p	192.168....	31...	5962929
o	192.168....	31...	6035711
d	192.168....	31...	6223775
f	192.168....	31...	6421725
b	192.168....	31...	6954942
u	192.168....	31...	7148201
v	192.168....	31...	7813652
u	192.168....	31...	8287889
w	192.168....	31...	8584105

```
(k,192.168.178.21,3100,...
(e,192.168.178.21,3100,1634
(o,192.168.178.21,3100,6035
(u,192.168.178.21,3100,8287
(w,192.168.178.21,3100,8584
```

Figure A.14.: Visualization of a broadcast in a random overlay network


Network Simulation

Options

[✓] Simulation Enabled

[✓] Use same values for outgoing and incoming traffic

Outgoing / Incoming traffic

Probability	0.86
Delay lower bound	1529
Delay Range	3412

Figure A.15.: Dialog for the manipulation of the network connectivity