Lookup in Peer-to-Peer Networks

Semester Thesis

presented by

Stefan Mehr

ETH Zurich, Switzerland

Supervisor:

Dipl.-Ing. Jan Mischke

Dipl.-Ing. ETH David Hausheer

Prof. Dr. Burkhard Stiller

of the

Computer Engineering and Networks Laboratory

June of 2003

Abstract

This paper describes the implementation of the AGILE algorithm that was created for this semester thesis. AGILE was developed by Jan Mischke and Burkhard Stiller and is described in detail in [1].

The algorithm can be used for efficient lookup of objects in peer-to-peer networks. Instead of flooding the whole neighborhood with messages as it is done in many implementations that are currently used, the number of contacted peers can be hold constantly small by routing the messages in a systematic way.

The underlaying principle of AGILE is similar to that of Pastry [2], Tapestry [3] or Chord [4] which all create an own identification space for peers and objects by using a specific hash function. Within this space, objects are linked at the nearest peer so that they can be found efficiently. “Nearest” refers to the hash values of the objects and the peers which are built of a sequence of bytes of a certain basis. Such hash values can be compared easily, e.g. by their big endian representation. Furthermore, AGILE works with semantic clustering which arranges peers and objects with similar interests close to each other, in a logical way. This is achieved by the creation of a hash value for the respective groups and their concatenation with those of peers or objects.

Routing the messages is done in the following way: Every peer sends a message to a peer that is situated nearer to the object, if that is not itself. For that purpose a peer must know a neighbor for every position and for every possible basis of its hash value, if such exists. As long as it is the nearest one itself it can proceed with the next digit. If it finally finds a nearer one it forwards the message to that neighbor. All those “nearer neighbors” are stored in a so-called “Routing Table”.

In the implementation, all information that is important for the algorithm is stored in a data storage module. This module consists (in addition to the routing table mentioned) of a Group-of-Interest (GoI) structure and a class for saving the links that point to the objects. The communication module is responsible for the data exchange between peers and abstracts generally of the type of transfer (the implementation uses UDP/IP as standard). Last but not least, a data processing module provides a correct application flow. It answers user input and handles all incoming requests by using the communication and the data storage modules and is seen as the core module of the application.

The implementation was inspected successfully by executing many tests in a small environment. Large tests could not be performed in the scope of this work because the decentralized execution of the program would need a distributed environment of huge scale. The realization of the AGILE algorithm clearly showed its practical adaptability which was the real goal of this semester thesis.

Kurzfassung

Dieses Dokument beschreibt die Implementierung des AGILE-Algorithmus, welche im Rahmen dieser Semesterarbeit erstellt wurde. AGILE wurde von Jan Mischke und Burkhard Stiller entwickelt und ist in [1] detailliert beschrieben.

Der Algorithmus wird benutzt, um in Peer-To-Peer Netzwerken Objekte effizient aufsuchen zu können. Statt die ganze Nachbarschaft mit Nachrichten zu überfluten, wie dies in vielen bestehenden Peer-To-Peer Netzwerken der Fall ist, kann durch gezieltes Leiten dieser Nachrichten die Anzahl der kontaktierten Peers konstant klein gehalten werden.

Das grundlegendes Prinzip von AGILE gleicht jenem von Pastry [2], Tapestry [3] oder Chord [4], welche mittels Hashing einen eigenen Identifikationsraum für Peers und Objekte erstellen. Innerhalb dieses Raums werden die Objekte beim nächstgelegenen Peer verlinkt, damit sie effizient gefunden werden können. “Nächstgelegen” bezieht sich auf die Hashwerte der Objekte und der Peers, welche aus einer Folge von Bytes einer bestimmten Basis bestehen. Solche Hashwerte können einfach miteinander verglichen werden, z.B. mittels Big-Endian Darstellung. Darüber hinaus arbeitet AGILE mit semantischen Gruppen, welche Peers und Objekte mit ähnlichen Interessen logisch gesehen näher zusammenbringen. Dies geschieht durch die Erstellung eines Hashwertes für die jeweilige Gruppe und deren Verknüpfung mit solchen eines Objektes oder eines Peers.

Das Leiten - oder Routen - der Nachrichten geschieht nach folgendem Konzpet: Ein jeder Peer schickt eine Nachricht, welche nicht für ihn bestimmt ist, an einen Peer weiter, welcher näher beim Objekt liegt. Dazu muss ein Peer auf jeder Position des eigenen Hashwertes für jede Basis mindestens einen Nachbarn kennen, sofern es einen solchen gibt. Dadurch kann er auf einer Position des Hashwertes leicht feststellen, ob er selbst oder ein anderer Peer der “nächste Nachbar” ist. Solange er selbst der Nächste ist, kann er mit der folgenden Ziffer fortfahren. Wenn er schliesslich einen näheren Peer findet, schickt er diesem die Nachricht weiter. Diese “nächsten Nachbarn” werden in einer sogenannten “Routing Tabelle” gespeichert.

In der Implementation werden alle für den Algorithmus relevanten Informationen in dem Datenspeicher-Modul abgelegt. Dieses enthält nebst der erwähnten Routing Tabelle auch eine Group-of-Interest (GoI) Strukur und eine Klasse für die Aufbewahrung von Links auf Objekte. Das Kommunikations-Modul ist für den Nachrichtenaustausch zwischen Peers verantwortlich und abstrahiert generell von der Art der Übertragung (wobei UDP/IP als Standard implementiert wurde). Schliesslich sorgt das Verarbeitungs-Modul für den korrekten Ablauf des Programms. Es bedient Benutzereingaben und erledigt Anfragen von Peers mit Hilfe des Kommunikations- und des Datenverarbeitungs-Modul und steht im eigentlichen Mittelpunkt der Applikation.

Die Implementation wurde durch mehrere Tests im kleinen Rahmen erfolgreich geprüft. Umfassende Tests konnten im Rahmen dieser Arbeit nicht durchgeführt werden weil die dezentralisierte Anwendung des Programms eine verteilte Umgebung von riesigem Ausmass erfordern würde. Die Umsetzung des AGILE-Algorithmus zeigte hingegen klar dessen praktische Anwendbarkeit, welche das eigentliche Ziel dieser Semesterarbeit darstellte.

Contents

1 Introduction

Peer-to-Peer (P2P) networks can be seen as high-complex distributed systems. Their complexity originates mainly from the inconvenience of searching objects within the whole network without wasting too much time and bandwidth ressources. Currently used networks work with simple search algorithms that do not care about those two requirements. New algorithms have been developed that improve both demands without devaluing the main subject, namely the search itself.

One of these algorithms called AGILE (Adaptive Group-of-Interest based Lookup Engine) has been chosen as subject for this thesis. Its concept is derived from similar ones like Pastry [2], Tapestry [3] or Chord [4], which all build a hash based identification space in order to distribute links that point to objects on peers. The links can be found through routing messages along paths that are given by the object’s and the peer’s hash values. Hash values can be calculated easily and obey a uniformed distribution of the base digits used at all positions and within all values. Therefore, the links will also be distributed uniformly inside the whole network. By keeping the invariant that a link is always located at the peer which has the nearest hash value in comparison with the object’s hash value (which is the destination of the link), every object can be found, or its absence can be evaluated, in a short amount of time and with few messages to be sent. The difference between AGILE and the other algorithms is the fact that every used hash value, either the peer ones or the object ones, are appended to the hash value of the Group-of-Interest (GoI) they belong to. The use of those GoIs effects a semantic clustering of objects and peers. Peers that perform many queries within the same GoI will become a member of that group and following queries will have to be forwarded to fewer hops than before.

It was the goal of this semester thesis to adapt, refine and improve the AGILE algorithm and finally, as the main part, try to create a working implementation. Some tasks of that implementation were selected as high priority, others as medium or low priority and some were optional to be performed. The implementation and the way it was developed are described in the following sections. All classes are described in detail and a conclusion evaluates the results of the thesis. Finally, a section describes proposals for future work with this implementation.

2 Implementation of AGILE

2.1 Problem solving approach

A deep understanding of the algorithm can only be reached by handling it over a long period of time. A way to achieve knowledge about it is to begin the implementation without caring about the details. The disadvantage of this strategy is that one eventually has to change the whole structure of the created work because of a new view of its concept. Nevertheless the strategy was chosen for this thesis.

Discussions about the communication modules were the main subject of the beginning work. These modules were planned to be used also by an implementation of another semester thesis, so they had to be designed very carefully in order not to have to be changed later. Because an important component had not been considered, namely the confirmation of packets sent, it finally had to be changed and does not coincide anymore with the other one. This causes no problems since the thought behind this concept was to safe work. Other design concepts were chosen during the implementation phase.

Some improvements and little corrections of the algorithm had been detected on implementing it, so the redesign occurred in a later state of work.

2.2 Solution of the given task

The creation of the implementation was started in a straightforward way. Some main modules were built in a first version and soon, the interaction between two copies of the compiled code could be tested.

Many little and some bigger problems occurred during the implementation. Most of these were detected at one of many testing phases that had been performed. But there was no problem with any promise to be solved. Everything could be handled, sometimes after the assimilation of the algorithm or reconstruction of whole code segments.

Finally, a well working implementation could be created, whose structure and modules will be described in detail in the following sections.

2.3 Class overview

The AGILE implementation can be structured into 4 important modules: user interface, communication, data processing and data storage (see Figure 1). Additionally, there are some helping classes (global classes).

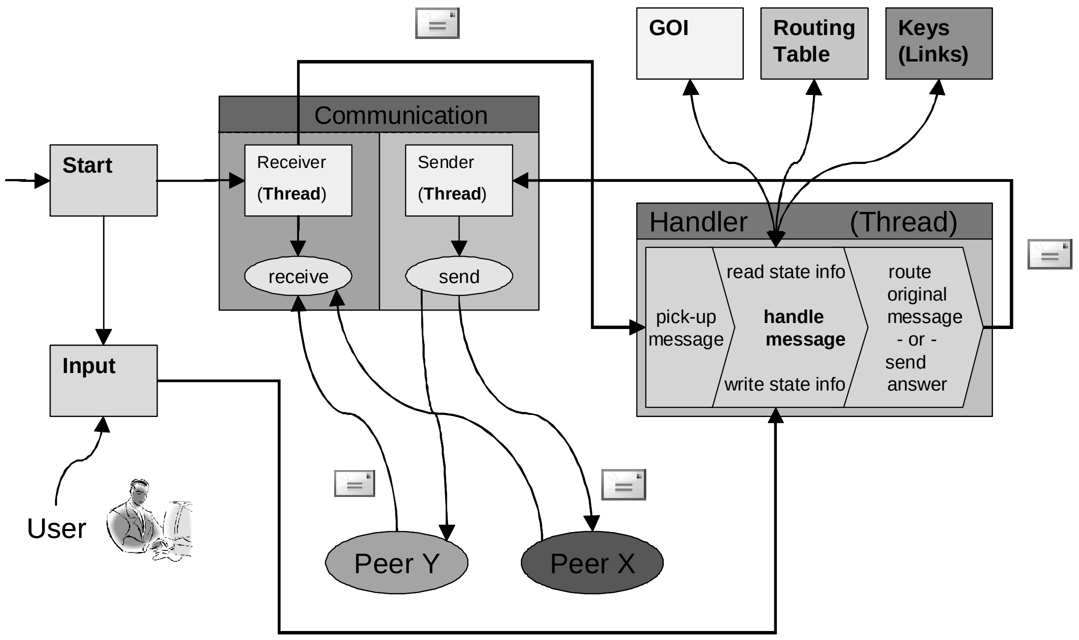


Figure 1: Important Modules and Classes of Implementation

The 'user interface' consists of the 'Start', 'Input' and 'Output' classes. The 'Start' class initializes the whole system. 'Output' contains all methods that print information to the standard output. The method 'waitForInput()' of the class 'Input' reads from the standard input, acts according to the type of the input, and loops until a 'quit' command is given.

The communication module is responsible for sending and receiving messages through UDP/IP packets. It consists of the classes 'Communication', 'Receiver', 'Sender' 'Waiter', 'Acknowledgements' and all message classes. At initialization, a 'Receiver' object is created and executed as a thread. Its 'run()' method calls the method 'receive()' from 'Communication' in a blocking way. When it receives a packet, it creates a new handler, passes the message and tries to receive the next one. The (blocking) 'receive()' method itself sends an acknowledgment to the sender or deletes acknowledgements (on receive) from the pending list (see '2.5.5 Acknowledgements class'). The message is usually sent by a 'Sender' thread, which calls the method 'send()' from 'Communication', or by the 'send()' method directly. This method sends a packet and waits until an acknowledgement is received or a time-out happened (see '2.5.4 Waiter class'), because of a crashing peer or a send failure. Because the 'Sender' is started as thread, the application is not blocked during the time-out period, and several failures on delivering messages to a crashing peer can be detected within one sole period.

The 'data processing' module consists mainly of the 'Handler' class which extends the 'Thread' class. A handler is instantiated for every task (incoming request) that has to be performed. That way, all tasks are independent of each other and the system will not be blocked by a single message, which could not be delivered (see '2.6.1 Handler class'). The instantiated handler picks up a message from the communication module and handles it depending on its type. On handling, it reads and updates information from the data storage module. That module defines mainly the process of the handled message, above all, the next peer, to which it has to be forwarded. As soon as the next peer is determined, the handler performs its last task: hand over the packet to the communication module for sending purpose. The ‘Algorithm’ and the ‘Hash’ class are also part of the ‘data processing’ module and are responsible for routing messages and calculating hash values.

The ‘data storage’ module consists of the classes ‘GOI’, ‘RoutingTable’, ‘Keys’, ‘OwnObjects’ and ‘QueryObject’. The ‘GOI’ class stores information (linking keys, routing table, level names and hash values) about every group of interest (GoI), to which the peer belongs. In this implementation, only one GoI is used, which is defined in the code and created at start time (see ‘Start class’ and ‘GOI class’). Every ‘GOI’ object contains one ‘RoutingTable’ object. This table stores ‘next neighbors’ of the peer (depending on its GoI). By routing a message, the table is used to find a specific ‘next neighbor’. At every field, several next neighbors are stored in order to provide fault tolerance. One GoI object holds also a list of ‘Key’ objects. A key is a link to an object of another peer. It consists of an ID and a list of hosts. All objects that are inserted in the network and whose hash value coincide with the ID of a key are linked by that key. If an object is hosted at several peers, all their addresses are stored in the linked list of the key. If someone searches an object that is linked at this peer, the addresses of all providing hosts are read from the key object and returned to the asking peer. The ‘OwnObjects’ class stores the addresses of the hosts which link an object that this peer is providing. A ‘QueryObject‘ object is used after sending queries into the network in order to store the answers.

All classes are described in more detail in the following sections.

2.4 User interface

2.4.1 Start class

The class ‘Start’ is the entry point into the implementation. Its ‘main()’ method is called on starting the application.

The first task it has to perform is checking the passed parameters. Secondly, it tries to initialize some important classes like ‘Constants’, ‘Hash’ and ‘Communication’, which depend either on those parameters, or on the system’s configuration. If one class fails to initialize, the agent is not able to work correctly and terminates by printing an error message to the standard output.

After initialization, a first predefined group of interest (GoI) is created. The routing table of this GoI is partially filled with the own address.

Other peers will not reach this one until it is inserted into the network. Therefore the ‘Receiver’ can be started without caring about incoming requests. It must in fact be started now because of the following request.

A (first) ‘insert_node_request’ is sent in order to insert the peer into the existing network. Destination of the message is a peer that must be known at startup (and is therefore passed as parameter). If this peer cannot be reached, a new virtual network is created. Otherwise, the peer will return a message containing all addresses of the hosts which are stored in its routing table at the current position (i.e. the whole row of the table - the current position is passed in the message), increase the position byte by 1 and forward it to the next hop, which is selected by the routing table. If the contacted peer contains an empty field in its table at the sending peer’s position, it stores its address, too. That way, some peers will include the new one into their tables and the new one will receive some answers.

The addresses included in these answers are put into the own routing table. After receiving these messages, the peer is considered to be inserted into the network (Consider Message Sequence chart in Figure 2).

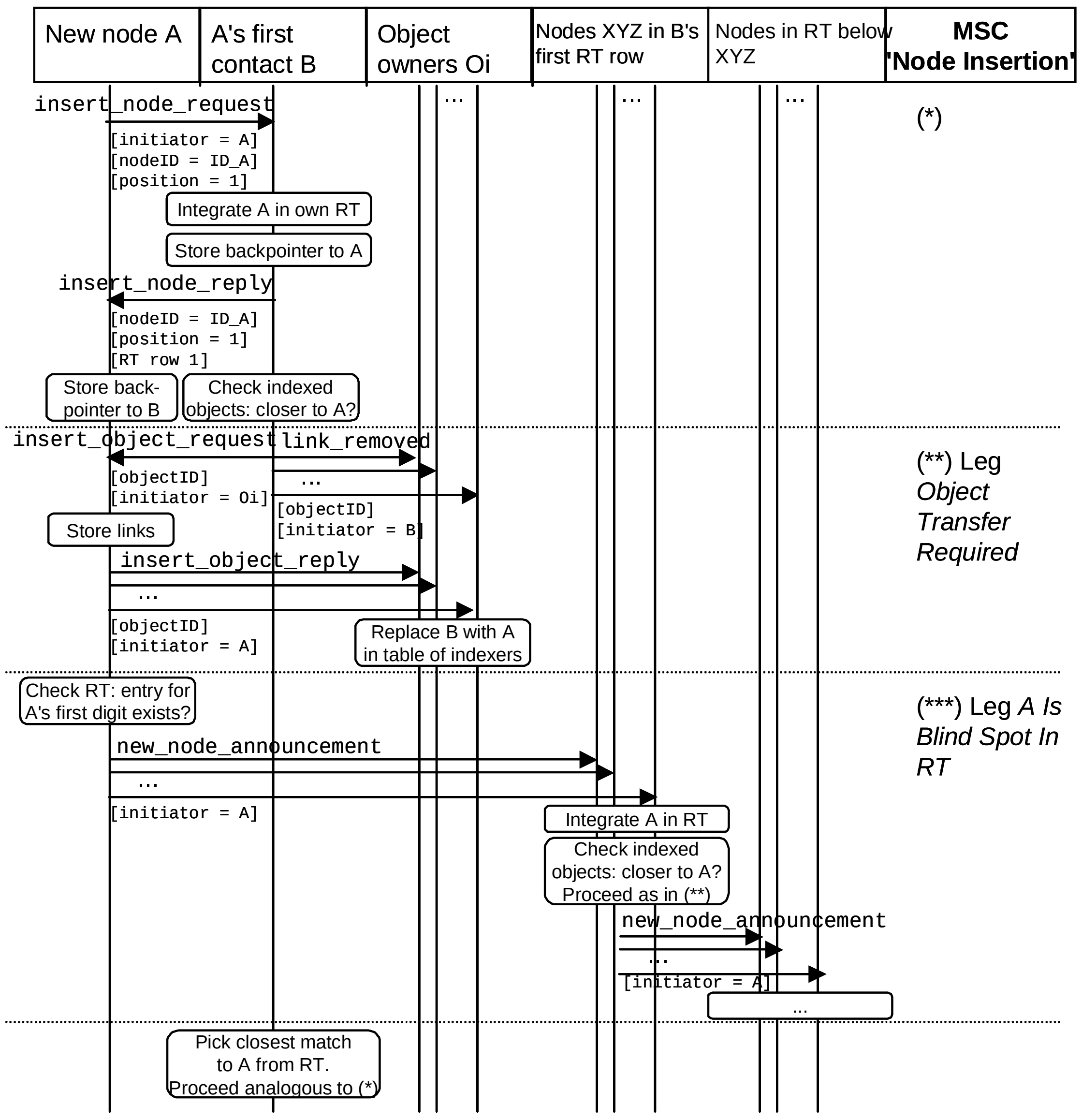


Figure 2: AGILE MSC Node Insertion [5]

The last step at startup is the insertion of all the objects the peers wants to share. In this implementation, the current directory is read (see 'Input class') and all files found are being hashed by their names and inserted into the network. For every insertion, an 'insert_object_request' is sent (see Figure 3). Every peer that receives an 'insert_object_request' forwards this message to the next peer on the path, or, if it is the nearest one, links the object and sends an answer to the requesting node. On receiving the answer, the peer updates its 'OwnObjects' object with the address of the indexing node (sender of the answer).

Finally, the 'waitForInput()' method from the Input class is called, which loops until the user quits the program.

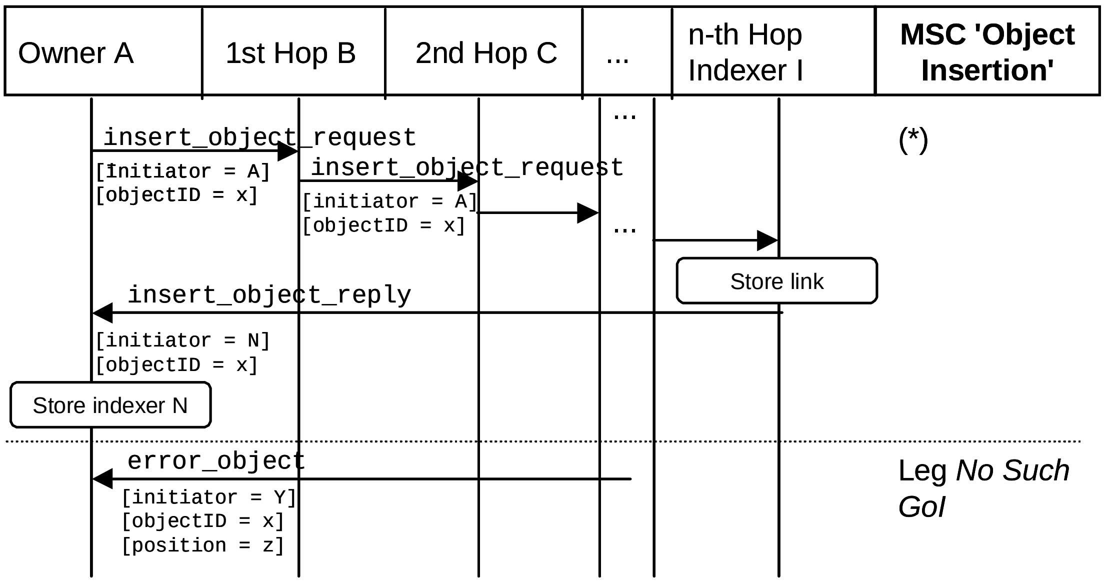


Figure 3: AGILE MSC Object Insertion

2.4.2 Input class

The standard input is handled by the only Input object, which was started by the Start class. The main method is called ‘waitForInput()’ and just loops until the user wants to quit. Users can perform searches, print routing tables, statistics, own object links and the links to other objects the peer holds. All printing methods are defined in the Output class. Output is always performed related to a specific GoI (except ‘print statistics’ which covers all GoI’s). There exists also a command to change this current GoI. The file search itself is handled in the loop, because it needs only few lines of code.

One method, called ‘getOwnObjects()’, is placed in this class because it inputs the files from the current directory at application startup.

When the user quits the application, some additional work has to be done: Remove the peer out of the network; let all peers which are linking your objects delete those links; tell all peers whose objects you are linking that you are leaving the network (some of this work can be improved (see ‘4 Summary, Conclusions, and Future Work’)). There are three more methods which perform these tasks. If a send request does not get an answer, the application will just continue, because it is going to quit. If another peer does not receive the ‘remove node’ request, it assumes that this peer is still living, until the other sends something to this. Then, the other will notice that the this one is dead.

2.4.3 Output class

All methods that are printing information to the standard output are defined in the Output class and are static. A very helpful method is called ‘printByteArray()’. It prints a byte array, usually a hash value, by outputting all bytes separated by a ‘:’ sign. This method is used for testing the program, i.e. comparing the hash values of objects with the ones of peers.

All other methods just print data like routing table content or linked keys to the standard output.

2.5 Communication module

2.5.1 Communication class

All Communication is done by the Communication class which operates with ‘UDP over IP’ packets and uses therefore ‘DatagramSocket’ and ‘DatagrammPacket’ objects. Only the first instantiation of an object initializes the sockets. The class consists mainly of a ‘send()’ and a ‘receive()’ method. A typical use of these two functions is as follows:

‘receive()’ was called on one peer (because every peer calls this method constantly) and blocks now until it receives a raw message. Another peer’s ‘send()’ method was invoked because the user wants to send a query. The method extends the message passed to it with a sequence number which is unique for one agent during a long period of time. It then puts this sequence number in an acknowledgement list (see 2.5.5 Acknowledgements class), sends the message to the peer mentioned above and waits for a short period of time. The receiving peer cuts off the sequence number from the message and instantly sends an acknowledgment to the sender, consisting only of this number and a normal header. Afterwards, it stores the message data in the destined byte array and returns the sender’s address to the calling module (‘Receiver’). The acknowledgment that has been sent is received by the message’s sender, which extracts the sequence number and tries to delete it from the acknowledgement list. If the ‘send()’ method called in the beginning hasn’t yet reached its time-out, it completes successfully and returns to its caller.

The ‘receive()’ method is blocking, which means that it waits until it receives something. It is called one time after another, just after ‘queuing’ the received message. The ‘send()’ method can be blocked during a pre-defined period of time (see ‘Constants class’). But the agent can send other messages at the same time in any case, because each call to ‘send()’ is normally done by a separate ‘Sender’ thread and every message contains its own ‘Communication’ object. Because of this fact, a crashed peer will never block another one. There exists a case, in which a handler sends several ‘insert_node_reply’s’ to the same peer at the same time. If the receiving peer crashed, all sending failures would be noticed after one instead of several amounts of time.

The sequence numbers for the acknowledgements as mentioned above can be obtained through a static synchronized method which ensures uniqueness. If the sequence number reaches its maximum value, it will be set back to 0. The probability of a collision is negligibly small.

2.5.2 Receiver class

A Receiver instance is created and started (because it is a thread) at application startup by the ‘main()’ method. After starting, it loops and performs three tasks: Receive data, create a message depending on the type of received data and start a new ‘Handler’ thread which will handle that message.

All data that is received must have a header of four bytes. The format of the header is displayed in Figure 4. Its first byte, the ‘Version’ byte, is always a ‘1’ in this implementation. The ‘Type’ byte describes the type of the message, which is defined in the Constants class. The ‘Body Length’ stores the length of the ‘Message Body’ which is the tail of the message. The smallest body length is 4 bytes (e.g. acknowledgement packets), the largest is defined in the Constants class. This value must be known, because one must pass it to the raw receive method provided by Java. The largest packet is an ‘insert_node_reply’, which contains a whole routing table row. One should take care about the ‘maxMsgSize’ constant if one adds new message types.

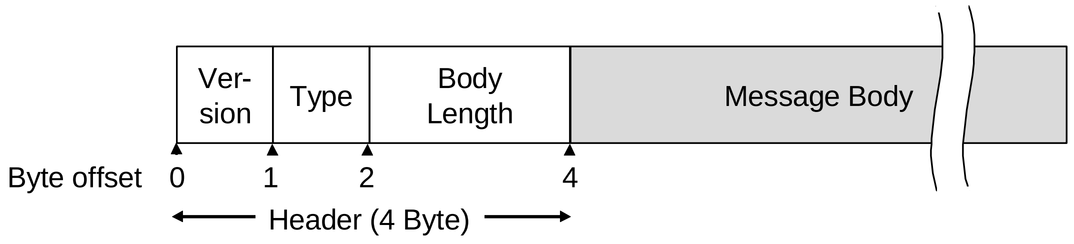


Figure 4: AGILE Message Format [5]

After extracting the header and the data, the method ‘handleEntry()’ is called which will create a new message depending on the type. Then, a new handler is generated, the message is passed as ‘Message’, and the handler is started (see 2.6.1 Handler class). Finally, the Receiver tries to receive the next message.

2.5.3 Sender class

Every instance of the ‘Sender’ class, which itself extends the ‘Thread’ class, can be run concurrently. The goal of this design is to achieve a non-blocking agent, which quickly detects a dead peer. Let us look at an example to explain this thought: Assumed a peer receives an ‘insert_node_request’ with the position byte equal ‘0’. As the sending peer uses the same GoI, its first 16 bytes of the node ID coincide with the ones of the receiving peer. In that case the latter one must send 16 ‘node_insert_reply’s’ to the requestor. For this purpose, it composes 16 messages and sends them, one after another. If the requestor dies unattended, all these messages will not be confirmed. Because they are all sent by the Sender thread, every Sender will recognize the death after the time-out period. If they were not concurrent, the sending peer would have to wait 16 times the time-out period. This schema is used by other messages, too.

The only thing a ‘Sender’ instance performs is trying to send the message several times, depending on its type. If still unsuccessful, a ‘routing table repair’ request could be sent (which is not implemented in this version; see code and ‘4 Summary, Conclusions, and Future Work’) or an output to the user could be effected.

2.5.4 Waiter class

The ‘Waiter’ class has to cover a small region of work. When the Communication’s send method is acting, a new waiter will be started, which will notify the sending thread after the time-out period has elapsed. If the receiving peer instantly sends an acknowledgement, the Waiter instance loses its attractiveness, because the sending thread has already been awoken.

This class could possibly be optimized, but because of lack of time and because the application was running correctly it was not changed (see ‘4 Summary, Conclusions, and Future Work’).

2.5.5 Acknowledgements class

The ‘Acknowledgements’ class extends the ‘LinkedList’ java class. An acknowledgement is put into and deleted out from the list by synchronized methods. The ‘waitForAck()’ method can be called to wait for a given acknowledgement number to be deleted from the list, which means the package with that number was delivered successfully.

2.5.6 Message classes

All message classes are contained in the messages package. Every message has its own class which is derived from the ‘Message’ class.

The ‘Message’ class contains all objects that every message will use (e.g. ‘version’, ‘type of message’). A special variable called ‘iAmNearestNeighbor’ indicates wether this peer is the nearest neighbor after the message has been routed (and therefore not forwarded!). Every message (or derived) object contains its own ‘Communication’ object, so that a message can be sent and confirmed independently.

Three functions are defined in the Message class: ‘send()’, ‘forward()’ and ‘route()’. ‘route()’ will use the Algorithm class to find the next neighbor. If that neighbor is not this peer itself, it forwards the message by the ‘forward()’ method. This method changes the ‘position’ byte of the message if existent (which is only the case in ‘insert_node_request’ and ‘insert_node_reply’), and sends it by the ‘send()’ method. As you can see, all those methods can end in a ‘send()’ call.

Every subtype of ‘Message’ contains two constructors: One for incoming and one for outgoing messages. The former one reads the data of the message and puts them into the appropriate variables. The latter one is called with all needed variable values as parameters and puts those together into the ‘data’ variable. Like this, all messages have their data stored in their variables plus in the special ‘data’ byte array. This ensures large freedom in treating a message as incoming or outgoing and profits on forwarding messages.

2.6 Data processing module

2.6.1 Handler class

The main part of the Handler class is the ‘run()’ method that is called by a ‘Receiver’ object. At start of this method, the ‘message’ variable has already been assigned and contains the received message. Depending on that messages type, the handler will cast and handle the message. The actions that depend on the types are now described in detail:

insert_node_request: All GoIs will be updated, regardless of the GoI of the message. If this step is skipped, new global GoIs cannot be inserted into the network! In all those GoIs, their hash codes are compared with the one of the message. For every position that coincides, an ‘insert_node_reply’ must be sent containing the whole row of the GoI’s routing table. Then, the message is routed by the main algorithm (see ‘2.6.2 Algorithm class’). Either, another peer must handle it or, if this peer is still the nearest neighbor, it must send one last ‘insert_node_reply’. Finally, the own GoI’s routing table must be updated and all objects (of that GoI) must be re-routed. The loop repeats now with the next GoI.

insert_node_reply: First, the GoI is searched which is the nearest one to the ‘nodeID’ value of the message. Then, its routing table is updated at the announced position and it is checked, if the message’s entry, where the own peer’s address must be stored, is empty. If so, a ‘new_node_announcement’ is sent to all peers in that row, including to itself (see below, ‘new_node_announcement’). This effects that all peers which are located on the same level or below the own peer’s know about its presence. Skipping this step would cause failures in the routing algorithm.

remove_node: The peer’s address are deleted from all GoI’s routing tables.

new_node_announcement: As above, the message is handled according to all GoI’s. First, the new peer’s address is added to the routing table. Then, all neighbors located below the indicated position are searched except the own peer itself and the message is forwarded to them with the position byte increased by ‘1’. Finally all links in this GoI are re-routed, because they could be closer now to the new peer’s hash code.

insert_object_request: The next hop is searched to forward the message, or, if this peer is the nearest neighbor, a link is created to that object and added to the GoI’s key list and an ‘insert_node_reply’ message is sent.

insert_node_reply: The key in OwnObjects is updated with the new indexing peer’s address.

error_object: An ‘error_object’ will be received if a ‘sent insert_object’ or ‘query’ belongs to a GoI that does not exist in the network. The only thing that happens is an output to the standard output object.

remove_link: The link from the appropriate GoI’s key list is removed.

link_removed: If a peer sends this message it must be leaving the network. In that case, its address is deleted out of the routing table (the peers nodeID will be calculated through its address). Then the peer’s own object which is not linked anymore is re-routed to another peer. If this peer is the nearest neighbor, it stores the link.

query: If this peer is the nearest one an answer is sent, otherwise the message is routed. The answer will contain a ‘0’ as ‘number of objects’ if no links of this ID were found, otherwise the real number of links and the addresses of all peers which are owning that object (consider Message Sequence Chart in Figure 5).

query_answer: If ‘number of objects’ is ‘0’, no peers were found which own that object. Anyway, the results of the query are printed to the standard output object.

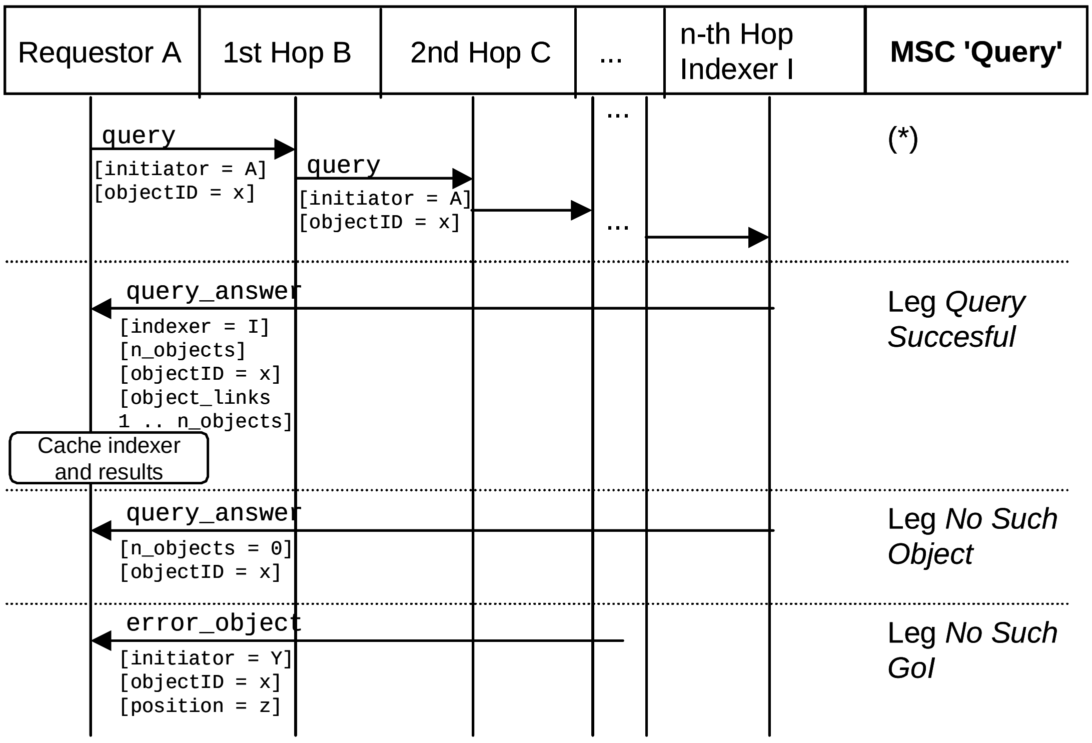


Figure 5: AGILE MSC Query [5]

2.6.2 Algorithm class

This implementation uses one main algorithm to find the next hop on a routing path to an object or a peer. That algorithm is defined in [1] and counts as the main part of this thesis. In this way, it gets its own class, named ‘Algorithm’. The only method it contains is the static ‘routeKey()’ method, which finds the next hop the passed message must be sent to. The algorithm first checks the GoI of the message and then its routing ID (which is either a node ID or an object ID). The first position of those two steps which does not coincide with the own node ID is authoritative for determining the next hop on the route. If no direct next hop can be found, a nearest neighbor must be evaluated that will act as next hop. There are also cases, mainly if the network consists of little peers, where the own peer is the nearest neighbor so that no message must be sent at that position. The variable iAmNearestNeighbor of the Message class will indicate that fact, and the module which called this method must react accordingly.

2.6.3 Hash class

The ‘Hash’ class is initialized during startup. This implementation uses the MD5 algorithm as hash function, but can be changed very simply to another Java supported function by changing the ‘Constants.hashFunctionName’ constant. It is recommended not to change it, because tests have proved its functionality.

The main function in this class is called ‘getHashValue()’. It can be invoked whit an input string and an integer value. The string is the one to be hashed, the integer defines the number of bytes that will be returned. Every level of the ID used for objects or peers has its own size, defined in the ‘Constants’ class. By hashing a part of the ID, this size must be passed to ‘getHashValue()’.

The method ‘getFileHash()’ calculates a hash value with all level names and the filename or the peer address as input values. It returns the whole ID of an object/peer.

2.7 Data storage module

2.7.1 GOI class

The ‘GOI’ class will be instantiated at least once (at startup) and every time a node adds itself to a new GoI. A static variable in this class stores a list of all GoIs a node supports. While handling requests, all GoIs are searched through this linked list. So, if a new GoI is created, it will be added to the list. There exists a static method that returns the first object in the list as well as a non-static method that returns the next object in the line after the current one.

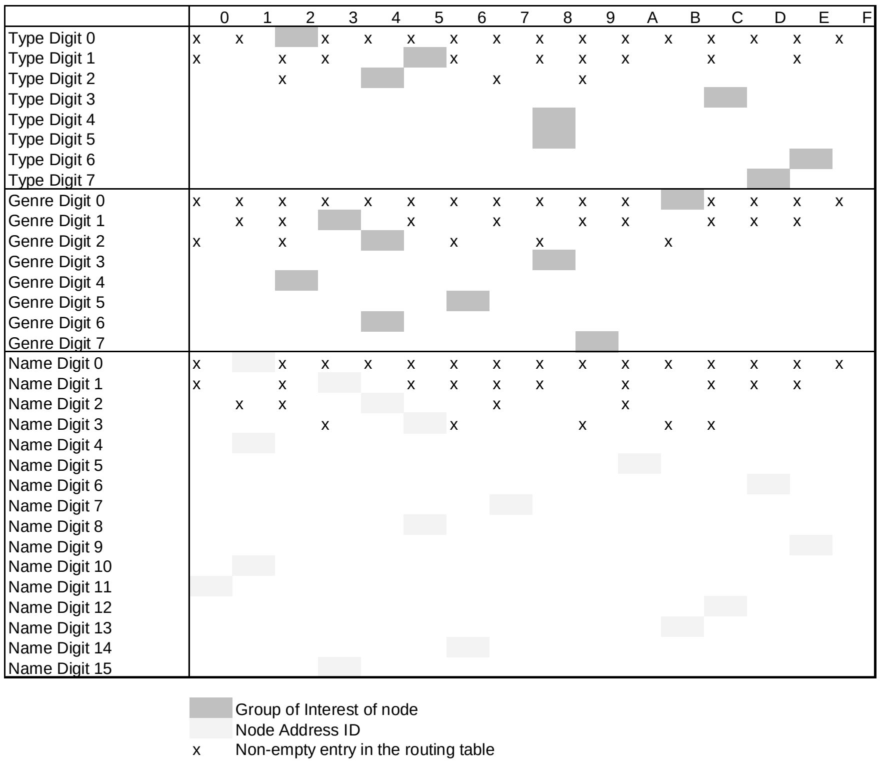
A ‘GOI’ object contains a routing table, a linked list of all keys it indexes and a string array containing the names of the levels of the appropriate GoI. To create a new GoI, the names of those levels must be defined.

The method ‘getGOI()’ returns a ‘GOI’ that belongs to the hash code passed. It is not used that often as the ‘findNearestGOI()’ method, which returns the ‘GOI’ whose hash code coincides most with the passed one. In order to finding a next hop of a message on the routing path, such a ‘nearest GOI’ must be found.

Objects (links) can be added to a ‘GOI’ through the ‘insertObject()’ method. The ‘keyID’ of the object as well as the peer which owns it must be passed as arguments. The method tests if the key exists already (because another peer can own the same object) and adds or extends the key and its hosting node’s address. In the same way, an object can be removed by ‘removeObject()’. Finally, there exist methods which return keys or addresses that belong to specific keys. These methods are mainly called by the handler in order to reply to queries.

2.7.2 RoutingTable class

A RoutingTable object is an important data structure which stores the addresses of a peer’s known neighbors in the network. Such a table is associated to a GoI and therefore initialized with every new ‘GOI’ instance. The table (see Figure 6) is two-dimensional, the horizontal one depends on the base digit of a hash value digit, the vertical one on the number of digits in a whole hash value. In this implementation, the constants that define the dimensions are set to 16 (base of digits) and 32 (number of digits) respectively. They can be changed in the ‘Constants’ class (see 2.7.1 GOI class). Each entry in this table stores a maximum of ‘Constants.nofNexthopsPerEntry’ next neighbors.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Type Digit 0	x	x	▨	x	x	x	x	x	x	x	x	x	x	x	x	x
Type Digit 1	x		x	x		▨	x		x	x	x		x		x	
Type Digit 2			x		▨			x		x						
Type Digit 3													▨			
Type Digit 4									▨							
Type Digit 5									▨							
Type Digit 6															▨	
Type Digit 7														▨		
Genre Digit 0	x	x	x	x	x	x	x	x	x	x	x	▨	x	x	x	x
Genre Digit 1		x	x	▨		x		x		x	x		x	x	x	
Genre Digit 2	x		x		▨		x		x			x				
Genre Digit 3									▨							
Genre Digit 4			▨													
Genre Digit 5							▨									
Genre Digit 6					▨											
Genre Digit 7										▨						
Name Digit 0	x	░	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Name Digit 1	x		x	░		x	x	x	x		x		x	x	x	
Name Digit 2		x	x		░			x			x					
Name Digit 3				x		░	x			x		x	x			
Name Digit 4		░														
Name Digit 5											░					
Name Digit 6														░		
Name Digit 7								░								
Name Digit 8						░										
Name Digit 9															░	
Name Digit 10		░														
Name Digit 11	░															
Name Digit 12													░			
Name Digit 13												░				
Name Digit 14							░									
Name Digit 15				░												

▨	Group of Interest of node
░	Node Address ID
x	Non-empty entry in the routing table

Figure 6: Illustrative AGILE Routing Table [5]

The meaning of an entry is as follows: The row signifies the number of digits that the stored peer’s hash code has in common with the one of the current peer (i.e. ‘number of the row’-1). The following digit of the peer’s hash code is then simply the column of the table.

The ‘RoutingTable’ class contains a ‘Level’ and an ‘Entry’ class (defined in the RoutingTable class itself). The Level class is just an object containing a description (e.g. ‘type’ or ‘genre’ or ‘domain’) and a hash code of that description. A ‘RoutingTable’ instance uses ‘Constants.nofLevels’ such ‘Level’ instances (including filename/address level). An ‘Entry’ object is also simple but each routing table contains many of them (e.g. 16*32=512 entries). It contains only an array of byte arrays which stores the next-hops. There are also functions to add, delete and return next-hops. The add function chooses a pseudo randomly next-hop it is going to replace, if no empty host exists and the own address is not present in the entry. The delete function deletes a destined hop whose address must be passed. ‘getNextHop()’

returns a non-empty next-hop or, if passed as parameter, a certain one. The third object a routing table contains is a hash code for the whole peer nodeID (which consists of its GoI and its address).

While creating a RoutingTable object, all objects are initialized and filled with appropriate values (by the method ‘setupRoutingTable()’). The only remaining method converts the entries of a given row into a byte array (so that they can be sent in a packet).

2.7.3 Key class

A ‘Key’ is a link to an object that is shared inside the network. Through a key, a user can find an object, because it stores the address of the peer that shares that object.

One ‘Key’ instance contains the ‘keyID’ of an object (which is the object’s hash value), a linked list of hosts that are sharing this object and, if the object is an own one, the filename of the object. The instance is included in the ‘keys’ list in every ‘GOI’ instance. If a peer has to save a link to an object, it creates a new key (which is a link) and adds it to the linked list of keys in the appropriate GoI (see 2.7.1 GOI class). If the key is already included in that list, because another peer owns an object with the same name, no new key has to be created but the peer’s address must be added to the host list in the existing key.

2.7.4 OwnObjects class

The ‘OwnObjects’ class is extended from the ‘LinkedList’ java class, which means that it can contain a variable number of objects (keys). All objects a peer wants to share must be added (i.e. their keys) to the only ‘OwnObjects’ instance that exists. The ‘Start’ class performs this task after sending the ‘insert_object_request’ of an object. If the peer gets an ‘error_object’ message for that object, because its GoI does not exists in the network, the handler will remove that object from the ‘OwnObjects’ instance. This instance can be accessed by the static method ‘getOwnObjects()’. Keys can be added, deleted and searched by an ID through their appropriated methods, which all are (and must be) synchronized.

2.7.5 QueryObject class

If a user performs a query, a new ‘QueryObject’ is instantiated and added to the static linked list ‘queries’ contained in the ‘QueryObject’ class. On receiving a successful answer, the peer adds all hosts contained in it to the ‘address’ array of links of the appropriate ‘QueryObject’ object. In that way, many queries can be performed after each other and answers will just be queued in the list. This class also supports methods to print the hosts that share the searched objects.

All performed queries are stored in that list, so if this implementation will be used by another application, it should be emptied and used in a suitable way.

2.8 Global classes

2.8.1 Methods class

In the ‘Methods’ class, there are some static methods defined, which will be called from different classes at different times and which are performing some general tasks. ‘append()’ will append bytes or byte arrays and return the concatenation of them. ‘compare()’ compares two byte arrays and tests them also for emptiness. Finally there are some functions that convert IP addresses, ports, byte arrays and numbers into each other.

2.8.2 Constants class

All global constants and application parameter derived variables are stored in the ‘Constants’ class. This class is divided into a ‘Communication’, a ‘Routing Table’, a ‘Messages’ and a ‘Hash Function’ part. The ‘Communication’ part defines IPs and ports of the current peer and its only know neighbor (at startup). The addresses are stored as objects, byte arrays and strings. Also, the communication time-out is set here. The ‘Routing Table’ part stores the dimension of the routing table which evolves from the sum of the size of all levels (which is the size of the ID of an object/peer; vertical dimension) and from the number of the base of a byte in the hash value (which must be a power of 2; horizontal dimension). These values can be changed, but the source code documentation should be considered carefully. Another constant defines the number of next-hops that will be stored in one routing table field. The ‘Messages’ part just maps the name of the messages to a byte value (e.g. ‘INSERT_OBJECT_REQUEST = 20’) and calculates the maximum size of a message which is needed in the receiving part of Communication. The last part, ‘Hash function’, only sets the used function and the number of bits that it returns (e.g. ‘MD5’; ‘128’).

2.8.3 Extern class

The Extern class is meant to be adapted if the implementation is to be used by another application. The ‘getLevel()’ method just returns a string depending on the passed level value.

3 Application usage description

3.1 System requirements

The implementation of the AGILE algorithm can be started on a computer that has a Java runtime environment installed, regardless of the operating system used. The tests have been performed with the Java version “1.4.0-b92” (on a Windows PC), therefore the application should be runnable with versions equal or higher than “1.4.0”. To test the version of the installed Java environment one should use the command “java -version” (see Figure 7).

```
C:\>java -version
java version "1.4.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-b92)
Java HotSpot(TM) Client VM (build 1.4.0-b92, mixed mode)
```

Figure 7: Java version verification

3.2 Program invocation

The application is a command line tool and can be started through the command in Figure 8 (replace the parameters with concrete values as shown in the example in Figure 9) if the current directory contains the file “AGILE.jar”.

```
java -cp AGILE.jar Start ‘port’ ‘neighbors_IP’ ‘neighbors_port’
```

Figure 8: Application invocation parameters

The attribute “-cp” defines the “JAR” file that contains the source code (which is here “AGILE.jar”). ‘Start’ is the name of the class that contains the ‘main()’ method. ‘Port’ is the UDP/IP port at which packets are received. Please note that also a second port is used with the value ‘Port’+1! Over that port the application sends its packets. Therefore, these two ports cannot be used by other applications or by other instances of AGILE, if several copies are launched on the same computer. ‘neighbors_IP’ and ‘neighbors_port’ are the appropriate values of a known neighbor.

```
java -cp AGILE.jar Start 1001 127.0.0.1 1000
```

Figure 9: Application invocation parameters (example 1)

If the agent is started twice, the IP and the port of the first one must be passed as parameters on calling the second one, so that they can interact (see Table 10). If a network is already built the agent can be started by passing some known IP/port combination of any host in that network. In the example of Figure 10, the first invocation does not find another peer (if none is started at port 1000). The second one will find a neighbor at port 1001. One is advised to use port numbers that differ in the last two digits because the output of the links will print only these two.

```
java -cp AGILE.jar Start 1001 127.0.0.1 1000
```

and - in another command line window -

```
java -cp AGILE.jar Start 2002 127.0.0.1 1001
```

Figure 10: Application invocation parameters (example 2)

After starting the first agent, the insert_node_request will not be successful, so a new network is built. This can be seen from the output which prints “No other node with same GOI found!“. The second agent in contrary will print “Node inserted!“.

Now, the application is ready for input. With the simple command ‘h’ (and ‘Enter’) the online help will be printed. All other commands are described below.

3.3 Input commands

The following input commands can be used to control the agent:

- ‘g’ - change GOI: Changes the current input/output GoI. Other commands (as print routing table) will always act on the current GoI. One has to input the level names of the desired, existing GoI (see Figure 11).

```
g
Input type: files
Input genre: documents
Current GOI: files; documents;
```

Figure 11: Change of Group-of-Interest

- ‘l’ - print links: Prints all links of the current GoI that the peer stores. It also prints the ID of the peer so that one can compare the hash values. The numbered links (e.g. ‘#1’) are printed with their hash values and followed by a list of all peers at which the objects can be found (short format, only the last two digits of the peer’s port number are printed). An example output can be seen in Figure 12.

```
l
/ links of group {'files','documents'}\
Node-ID: 3:4:4:0:13:3:1:14:1:6:3:10:4:13:5:15
link #0: 7:7:13:2:8:9:6:12:3:14:11:5:4:4:5:4 - 01 02
link #1: 14:11:5:11:10:6:0:10:4:15:14:11:1:13:12:3 - 01 02
```

Figure 12: List of links

- ‘m’ - print small routing table: Prints the routing table of the current GoI in a small format. Every line represents the row of the appropriate position in the routing table. An ‘X’ denotes that the entry in the given column is non-empty in contrast to a ‘-’ (see Figure 13).
- ‘o’ - print own files: Prints the hash values and the addresses of the peers where the links of all own objects are stored (see Figure 14).
- ‘q’ - quit program: Ends the application

```
m
----X-----------
-----X----------
------X---------
---X------------
----------X-----
```

Figure 13: Small routing table

```
o
/ where my files are linked at \
my address: 129.132.57.68@1001
7:7:13:2:8:9:6:12:3:14:11:5:4:4:5:4-129.132.57.68@2002
14:11:5:11:10:6:0:10:4:15:14:11:1:13:12:3-129.132.57.68@2002
```

Figure 14: Links to own files

- ‘r’ - print routing table: Prints the routing table of the current GoI (only one address per entry is shown). If an entry’s first hop stores an address, that address is printed, otherwise a ‘-’ is printed. Above that table an overview of the levels is printed (see Figure 15).

```
r
Levels:
files - 4:5:11:9:6:3:3:9
documents - 2:1:15:6:4:13:10:1
129.132.57.682002 - 3:4:4:0:13:3:1:14:1:6:3:10:4:13:5:15

Entries:
-       -       -       -       129.132.57.68:2002      -       -
-       -       -       -       -       -       -       -
-       -       -       -       -       129.132.57.68:2002      -
```

Figure 15: Large routing table

- ‘s’ - search a file: Searches a file with the GoI and the name one has to input (see Figure 16). After searching the result is printed (the peers where the objects can be found or an error message).

```
s
Input type: files
Input genre: documents
Input filename: AGILE.jar
search: [AGILE.jar] in files, documents,..
AGILE.jar is hosted at:
129.132.57.68@2002; 129.132.57.68@1001;
```

Figure 16: Results of a file search

- ‘t’ - print statistics: Prints all GoI’s with their names an the number of all files to which it stores a link (see Figure 17). The number in parenthesis denotes the number of unique object names.

```
t
hosted links of group {'files','documents'}:
14 (7 different)
```

Figure 17: Statistics of a Group-of-Interest

- ‘y’ - print performed queries: Prints all queries one has performed (see Figure 18).

```
y
AGILE.jar is hosted at:
129.132.57.68@2002; 129.132.57.68@1001;
```

Figure 18: List of performed queries

The application reads the filenames of the system’s current directory. If one tries to (successfully) search a file, the GoI should be “files - documents” because it is the one that every peer belongs to and the name should belong to a file in the current directory. The file “AGILE.jar” (note lowercase/uppercase) will normally be such a filename.

4 Summary, Conclusions, and Future Work

4.1 Briefing

In the scope of this work, an implementation of the AGILE algorithm was built and successfully tested. All high and some medium priority goals of the thesis could be attained, but not all optional tasks could be performed. The main goal, namely demonstrating the practical adaptability of the algorithm, was achieved by the implementation and the inspection of a correct application and data flow.

4.2 Conclusions

Large tests of the created implementation could not be performed because of lack of time and absence of a suitable test-environment. Nevertheless the interaction between some running copies was tested. It can be said that the algorithm works correctly in a small distributed environment. It had to be changed a little bit in order to satisfy all imaginable cases. During the tests there were no indications that the algorithm would not work in a large environment, whereas the implementation itself is limited in the number of requests it can handle.

Some tasks as enhanced fault tolerance or code improvement can be done in a future version. The important ones are described below in short manner.

4.3 Proposals for future work

Some code segments are not programmed fully optimized and could be improved. Four concrete affected sections are the following:

- A routingID should be used instead of both nodeID and objectID in a message object
- The ‘goi’ variable should be set on creating a new message (incoming or outgoing) instead of assigning it in the handler thread.
- The ‘Communication’ class could be redesigned and the ‘Waiter’ class deleted and substituted.
- Less threads should be used or limited by a ‘maxiumum_threads’ variable.

In a large distributed environment fault tolerance is a very important task. If a peer crashes or is taken out of the network before terminating, the other peers are not informed of its absence. The links they are holding point to a non-existing peer and the links, the dead peer was holding are not stored at its nearest neighbor. Some additional actions would increase the availability of the links:

- Object link redundancy: Every object that is inserted into the network is also inserted with an added “salt value” (e.g. ‘filename_1’ instead of ‘filename’). If an object is not found by its name, it can be found by its other values.
- Report old links: If a peer gets a query_answer and send a download request to peer, that does not exist any more, it should report the dead link to its indexer. This can simply be done by a ‘remove_object’ request.

- Routing table repair: If a peer does not receive any answer to all requests it sent to its neighbors (which are all located in the same routing table entry), it must perform a routing table repair. Therefore it can contact all nodes at the same level or all nodes that are located below itself. They should have stored addresses of peers that belong to that routing table entry. If they are different, they should be sent to the requesting peer. For this purpose, a new message should be used (e.g. 'routing_table_repair_request(initiator | partial_ID)').
- Extended remove_node message: A peer that receives a remove_node message forwards it to all peers below itself. In that way all important peers can be contacted. Another possibility is to save "backpointers". A peer stores all addresses of other peers that are holding them in their routing tables. These backpointers must be created at every insert_node_request and every new_node_announcement message.

Simulation of the application in a large, distributed environment would prove (or disprove) the practical functionality of the algorithm in a bigger scope. An integration into a file sharing agent and a performance analysis would also show is possible usability as a commercial product.

5 References

[1] J. Mischke, B. Stiller: *Peer-to-peer Overlay Network Management Through AGILE: An Adaptive, Group-of-Interest-based Lookup Engine* (Extended Version); ETH Zürich, Switzerland, TIK Report Nr. 149, August 2002.

[2] Druschel, Rowstron: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*; IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, 2001, pp. 329-350.

[3] B. Zhao, J. Kubiatowicz, A. Joseph: *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*; Technical Report UCB/CSD-01-1141, Computer Science Division, U.C. Berkeley, April 2001.

[4] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris, I. Stoica: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*; ACM SIGCOMM, San Diego, August 27-31, 2001.

[5] J. Mischke: *AGILE Specification*, ETH Zürich, Switzerland, Unpublished Draft.

6 List of Figures