Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Pascal Stucki

Analyzing MorphMix, a Peer-to-Peer based System for Anonymous Internet Access

Student Thesis SA-2003.12
Winter Term 2002/2003

Tutors: Marc Rennhard, Matthias Bossardt
Supervisor: Bernhard Plattner

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Abstract

MorphMix is a network protocol that offers anonymous data traffic to its users. It is based on the idea of a network of so called MIXes, which receive the data of other users and forward them to other MIXes (and finally to the data's original receiver). In contrary to traditional MIX-networks, MorphMix is designed as a peer-to-peer protocol where every user plays the part of a MIX, so using the application to anonymously request its own pages as well as forwarding other users requests. P2P-MIX-networks solve most of the problems of traditional networks: they offer a better scalability and minimize the danger of traffic analysis with pattern matching or legal attacks against the providers of the static MIXes. But dynamic P2P networks lead to new problems: dynamic MIXes may leave the network at any time and by this interupt used connections. Attackers have the possibility to maintain (several) malicious MIXes to analyze net traffic or even forwarding wrong data (or no data at all).

The task of this student thesis is to write a simulator to test the stability and performance of the MorphMix protocol. The desired number of nodes (>10000) the application should be able to simulate make field trials too expensive. Already existing simulators implement all details of the underlying network protocols (TCP/IP, UDP) and are therefore too slow for bigger simulations. The code is written in Java and documented with JavaDoc, making it easier for other people to extend the simulation with new details. The object oriented design of the network environment includes classes for nodes, links and tunnels. The transferred data is represented by instances of the classes Message and Packet. The whole simulation is based on events that are managed in chronological order by priority queues. An occurring event (e.g. "send 20 bytes") will produce some later events concerning other nodes (e.g. "receive 20 bytes").

The application includes an adjustable logging mechanism, which stores the most important events in a log file for further analysis.

The simulated network environment is described with parameters stored in a properties file, which allows to change the simulated environment and the protocol parameters without changin ga single line of code.

Last but not least the tool includes several mechanisms to analyze the generated log files and produce statistical results about the simulation's behavior.

After some performance tuning the simulator is able to simulate the requested number of nodes (10'000) in real time or even faster. the simulation has been tested again and again during the development process. The produced results seem plausible and consistent. The simulator is in a stable and well documented status and can now be used to test different aspects of the MorphMix protocol.

Zusammenfassung

MorphMix ist ein Netzwerkprotokol, da es Benutzern erlaubt, anonyme Datentransfer zu tätigen. Es basiert auf dem Prinzip eines Netzwerkes mit sogenannten MIXes, welche die Daten anderer Nutzer entgegennehmen und an andere MIXes (und schlussendlich an den eigentlichen Empfänger) weiterleiten. Im Gegensatz zu traditionellen MIX-Netzwerken ist MorphMix als peer-to-peer-Protokol geplant. Jeder Benutzer übernimmt glleichzeitig die Aufgaben eines MIXes und leitet die Daten anderer Benutzer weiter. Der P2P-Ansatz löst viele Probleme traditioneller MIX-Netzwerke, etwa die schlechte skalierbarkeit der Anzahl MIXes und die Gefahr von Traffic Analysen mittels Pattern Matching oder legalen attacken auf die Betreiber von MIX-Knoten. Auf der anderen Seite ergeben sich aber durch die Dynamik des Netzwerkes auch neue Probleme. MIXes können sich jederzeit aus dem Netzwerk verabschieden und laufende Verbindungen unterbrechen. Auch bietet sich für Angreifer die Möglichkeit, (mehrere) "bösartige" MIXes zu betreiben, die die weiterzuleitenden Daten zu analysieren versuchen oder auch falsche oder gar keine Daten weiterleiten.
Um die Stabilität und Performance von MorphMix zu testen, sollte als Semesterarbeit ein Simulator erstellt werden. Ein Feldversuch kan wegen der gewünschten Anzahl simulierbarer Knoten (>10000) nicht in Frage, bereits existierende Simulatoren implementierten die Netzwerkprotokolle (TCP, IP, UDP) zu detailiert und erwiesen sich deshalb als zu langsam. Da der Simulator nach Beendigung erweitert werden sollte, wurde als Programmiersprache Java gewählt und der ganze Code mit JavaDoc ausführlich kommentiert. Die Netzwerkumgebung wurde objectorientiert mit Nodes, Links und Tunnels modelliert, Daten werden mit Messages und Paketen simuliert. Die Simulation wird mit events gesteuert, die in chronologischer reihenfolge in PriorityQueues verwaltet werden. Ein auftretender Event "sende 20 Bytes") löst meist weitere, zeitlich spätere Events aus (z.B. "empfange 20 Bytes").
Die zu simulierende Netzwerkumgebung wird mit in einem Properties File abgelegten Parametern beschrieben. Dieser Mechanismus erlaubt es, die simulierte Umgebung und Protokoleinstellungen anzupassen, ohne eine einzige Zeile Code verändern zu müssen.
Die entwickelte Applikation beinhaltet einen einstellbaren Logmechanismus, der die wichtigsten (gewünschten) eregnisse in einem Logfile abspeichert.
Ausserdem verfügt die Applikation über verschiedenste Tools zur Analyse erstellter Logfiles. Diese erstellen aus den abgespeicherten Log-Nachrichten statistische Daten über das Verhalten der Simulation. Analysiert werden kann etwa die Grösse der versandten Daten oder die benötigte Zeit, um eine Seite zu laden.
Nach einigem Performance Tuning ist der Simulator in der Lage, the angeforderte Anzahl Knoten in Echtzeit oder sogar noch schneller zu simulieren. Parallel zur laufenden Entwicklung wurde der Simulator immer wieder diversen Tests ausgesetzt. Die erhaltenen Testresultate erscheinen plausibel und in sich konsistent. Die Applikation befindet sich nun in einem stabilen und gut dokumentierten Zustand, der es zukünftigen Entwicklern leichter machen sollte, die Anwendung weiter zu entwickeln und den Simulator zu Testzwecken einzusetzen.

Contents

1 Introduction 6
1.1 Motivation . . . 7
1.2 Outlook . . . 7

2 Problem 8

3 Design 9
3.1 Programming Language . . . 9
3.2 Class Design . . . 9
3.3 Event driven solution . . . 9

4 Initializing the Simulator 10
4.1 Specifying the Test Environment . . . 10
4.1.1 Initialize Properties Reader . . . 10
4.1.2 Initialize Simulator . . . 10
4.1.3 Initialize Logger . . . 10
4.1.4 Initialize Random Number Generator . . . 10
4.2 Generate Network Environment . . . 11
4.2.1 Generate Web Server . . . 11
4.2.2 Generate Nodes . . . 11
4.2.3 Generate Links to Web Server . . . 12
4.2.4 Generate Links to other Nodes . . . 13
4.2.5 Generate Tunnels through Network . . . 14
4.2.6 Generate TunnelSteps . . . 14

5 Simulation: Iterating in Rounds over all Nodes 16
5.1 Simulate Web Server . . . 16
5.2 Loop over all Nodes . . . 16
5.2.1 Update Links . . . 16
5.2.2 Set Node Offline . . . 16
5.2.3 Start Browsing . . . 16
5.2.4 Stop Browsing . . . 17
5.2.5 Check for Node Events . . . 17
5.2.6 Simulate Sending for Node . . . 17
5.2.7 Simulate Receiving for Node . . . 17
5.2.8 Check Tunnels For Timeouts . . . 17
5.3 Print simulator status . . . 18

6 Start Web Request 19
6.1 Generate Web Request Message . . . 19
6.1.1 Generate Packets . . . 19
6.2 Send Web Request . . . 19

7 Simulate Sending 20
7.1 Distribute Bandwidth among Links . . . 20
7.2 Distribute Bandwidth among Tunnels . . . 20
7.3 Flow Control . . . 21
7.3.1 TCP window flow control . . . 21
7.3.2 Credit based flow control . . . 21
7.4 Simulate Sending of some Bytes . . . 21
7.5 Re-share unused Bandwidth . . . 22

8 Simulate Receiving 23
8.1 Distribute Bandwidth among Links . . . 23
8.2 Distribute Bandwidth among Tunnels . . . 23
8.3 Simulate receiving of some Bytes . . . 23
8.4 Receive a whole Packet . . . 23
8.5 Receive a whole Message . . . 24
8.5.1 Message reached Web Server . . . 24
8.5.2 Message reached Requestor . . . 24
8.6 Re-share unused Bandwidth . . . 25

9 Using properties files to set the test environment 26
9.1 The Test Case Properties File "WebRequest.properties" . . . 26
9.1.1 The properties used for setting the simulated network environment . . . 26
9.2 Distributions . . . 29
9.2.1 Fixed Value . . . 29
9.2.2 Random Distributed Property with Minimal and Maximal Value . . . 29
9.2.3 Property with several possible Values and their Probabilities . . . 30
9.2.4 Pareto Distribution for Number of Embedded Objects in an Index File . . . 30
9.2.5 Pareto Distribution for File Size . . . 30
9.2.6 Browse Delay . . . 30
9.3 The logging properties file . . . 30

10 The logging mechanism 31

11 Analysis 32
11.1 Status Analysis . . . 32
11.2 Uptime Pattern Analysis . . . 32
11.3 Message Analysis . . . 32
11.4 Failure Analyzer . . . 32

12 Testing 33
12.1 Analyzing one Request . . . 33
12.2 Correctness Distributions . . . 36
12.3 Plausibility tests for number of online, active, browsing, downloading nodes . . . 36
12.4 Plausibility tests for needed Time . . . 37
12.5 Plausibility tests for sizes . . . 38

13 Conclusions 41

14 Acknowledgements 42

A Timetable 43

B How to use the MorphMix Simulator 44
B.1 Requirements . . . 44
B.1.1 Simulator . . . 44
B.1.2 Java Standard Development Kit . . . 44
B.1.3 Zip Tool . . . 44
B.2 Installing the Simulator . . . 44
B.2.1 Unzipping the file Simulator.zip . . . 44
B.3 Compile the Java Files . . . 44
B.4 Generate a Test Environment . . . 44
B.5 Start the Simulator . . . 45
B.6 Analyze the Log File . . . 45

List of Figures

1 Traditional MIX Network with static MIXes 6
2 Peer-to-Peer MIX Network with static MIXes 7
3 Unconnected Network with Server and Nodes 12
4 Network with Connections to the Server 13
5 Network with Connections between Nodes 13
6 Tunnel from Node 1 to the Server 14
7 Tunnel from Node 1 to the Server in Detail 15
8 Bandwidth Sharing among active Links 20
9 Bandwidth Sharing among active Tunnels 21
10 Number of nodes included in the network with fixed value 29
11 Link Delay with Min and Max 29
12 Link Bandwidth with three possible values and their probabilities 30

List of Tables

1 Results Node Analysis 36
2 Results Time Analysis 38
3 Results Size Analysis 38
4 Timetable 43

1 Introduction

Towards the end of the 20th century, the world wide web became an alternative to the traditional communication and information media. All these media provide a certain degree of anonymity: a caller may hide his phone number, the reader of an article remains anonymous and so does the consumer of a TV show. But today, an Internet user leaves a lot of traces on his way through the net: the data entered in input fields, his IP address, used software and settings. For most users, the loss of anonymity results in personalized spam mails and stored data about his interests, but there are countries were the access to political or religious web sites is impossible or forbidden. It is therefore desirable that the Internet offers anonymity as well. The underlying protocols do nothing to hide the identity of the communicating parties, but there are more and more applications ready to fill the gap. Encryption can be used to shield the transmitted data from third parties, but the IP header of the packets are still transmitted in the clear and give away the identities of the sender and receiver. Another problem encryption doesn't solve is the anonymity of the sender towards the recipient. Sender anonymity is required by most Internet activities such as anonymous web browsing or pseudonymous e-commerce. It is therefore desirable that the underlying network provides anonymity. Therefore the data arriving at the recipient should not contain any information about the sender's identity. It must be impossible to trace back the data's way through the net.
The simplest solution offered are proxies or anonymizers. These are applications or web pages were the user may enter the requested URL and the proxy requests the web page in its own name and forwards it to the user. But the user's security depends on its confidence in the proxy and pattern analysis attacks may break the anonymity.
The most prominent approach to achieve a relatively high level of anonymity are MIX networks [1]. Traditional MIX-based systems consist of a static, well-known set of highly reliable MIXes. A small traditional MIX network is shown in figure 1. As there are only few MIXes, traffic analysis may be used to match incoming and outgoing packets at every MIX. To be resistant to traffic analysis attacks at a MIX, cover traffic must be used, which results in significant bandwidth overhead. To counter end-to-end traffic analysis attacks, a user must be sending and receiving a lot of packets at a time, which results in even more cover traffic. Additionally, static MIX networks suffer from scalability problems, as the number of MIXes should grow with the number of users. Finally, as there are only few MIXes, their operators could be targeted by legal attacks.

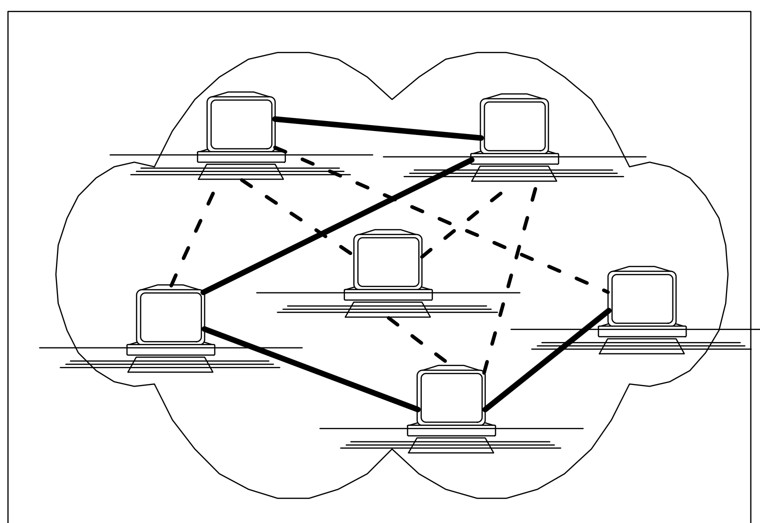

Figure 1: Traditional MIX Network with static MIXes

MorphMix is an alternative approach to achieve anonymity in the Internet. Each MorphMix node is a MIX and anyone can easily join the system. A small peer-to-peer MIX network is shown in figure 2. The significantly increased number of MIXes makes traffic analysis attacks a lot more

complicated and expensive. In particular, the drawback of cover traffic can be avoided, as the data each node is forwarding for other nodes are sufficient to shield its own requests [3].

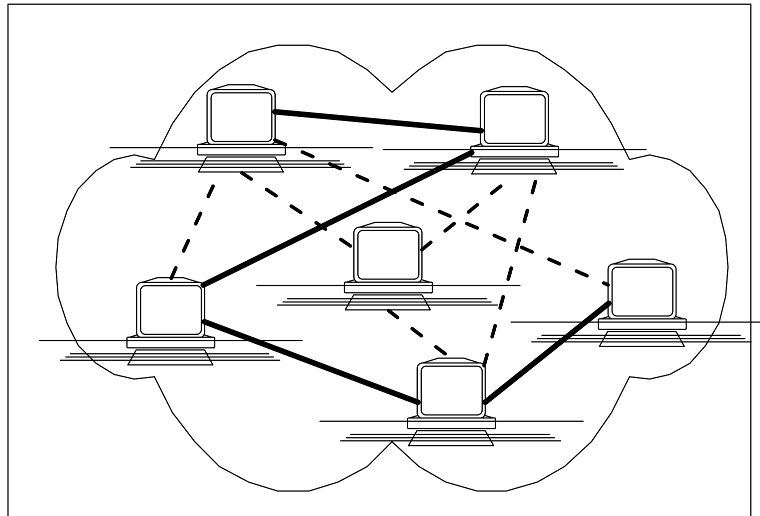

Figure 2: Peer-to-Peer MIX Network with static MIXes

But the opportunity for anyone to run a MIX also introduces new challenges. An adversary can operate several malicious nodes in the system and try to break the anonymity of other users by getting full control over their paths. To counter this attack, MorphMix employs a collusion detection mechanism to identify compromised paths with high probability. Another uncertainty is the performance of MorphMix concerning the dynamic joining and leaving of new MIXes.

1.1 Motivation

To find out more about the stability of end-to-end connections and the performance of the system under specific circumstances, a simulator is needed. A field trial is too expensive because MorphMix is supposed to handle a large number of nodes. Existing network simulators like JNS[1] or JXTA[2] simulate the underlying internet protocols (mostly TCP/IP) in great detail and their performance isn't good enough to simulate such a large number of nodes over a long period. To test the performance and stability of the MorphMix network it is therefore necessary to design and implement a less detailed and faster network simulator.

1.2 Outlook

This thesis covers the following aspects: Chapter 2 describes the goals of the student thesis. Chapter 3 offers an overview over the chosen design. The following chapters explain the simulator's functionality in more details, starting with its initialization (chapter 4), the simulation of one round (chapter 5), the generation of a new web request (chapter 6) and the process of sending and receiving data (chapter 7 and 8). Chapter 9 explains the usage of the properties files and its entries, chapter 10 covers the functionality of the logging mechanism. The included logfile analysis tools are explained in chapter 11. The report includes some test results and plausibility checks as well (chapter 12). It ends with some conclusions (chapter 13). A timetable and a manual for the MorphMix simulator can be found in the appendix.

[1] http://jns.sourceforge.net
[2] http://www.jxta.org

2 Problem

The goal of this semester thesis is to design and implement a simulation tool to analyze MorphMix. The simulator should be kept simple enough to give quantitative results about the performance of MorphMix for up to 10'000 nodes. But the tool should not be too simplistic in a way that the results have little meaning for a real-world environment.
The creation of a new simulation tool includes the following steps:

- Get more information about the project:
  - Study the MorphMix system by reading the technical report[2]. It is important to understand the ideas behind the chosen design so that they can be included in the simulation.
  - Analyze already existing network simulators such as JNS or JXDA and find reusable design ideas.
- Create the design for the simulator:
  - Decide which parts of the MorphMix network design should be included in the simulator. Two aspects are important for the decision about whether a detail should be included or not: how will its implementation help to analyze the MorphMix network, and how much will it slowdown the performance of the simulation. MorphMix should be able to handle a large number of nodes, implementing all protocol aspects would require far to many resources and running time. On the other side, the results of a too simplistic simulator have only little meaning for a real-world environment. The task is too find the balance between the needed performance and the exactness of the simulation.
  - Design the simulator: After having decided about the functionality of the simulator, a simple, useable and extendable design has to be chosen. This point includes decisions about the used classes and their functions.
- Implementation:
  - It is important to write extendable and well readable code so that the simulator can be further implemented by other developers. The Java code is following the standard Java programming rules and paradigmas and is commented with JavaDoc. Further information about the chosen design can be found in chapter 3.
- Testing:
  - The simulator has to be tested for its correctness. It must fulfill the performance requirements and generate plausible results.

3 Design

3.1 Programming Language

The given task allowed the simulation to be written either in C or Java. The decision for the object-oriented approach was taken because of the author's better knowledge and experience in this language, the existence of other, more detailed simulation tools with published source code written in Java and the desired extendibility of the program. The code is documented with the standard documentation tool for Java, JavaDoc, and is further explained in this report.

3.2 Class Design

As the simulator must be fast enough to run simulations with up to 100'000 nodes, not all the details of the MorphMix protocol could be implemented. On the other side, the simulator should be useable for different testing purposes and produce significant results. The decision to omit a detail of the protocol doesn't mean that it will not be implemented later on. All objects of the network are represented by instances of Java classes. The users running the application are simulated by instances of the class "Node". The Class "Link" represents the connection between two nodes. If a node wants to send some data, it uses one of its "Tunnels", which represent an end-to-end connection between sender and receiver going over several MIXes. A tunnel consists of several "TunnelSteps", which is one of the links the tunnel consists of. The data sent over the network is represented by the classes "Message" and "Packet". All mentioned classes are further described in the following chapters.

3.3 Event driven solution

All the actions occurring during the simulation are represented by time based events. As the simulator should be able to simulate several thousand nodes, it was necessary to manage the thousands of events efficiently. The events are stored chronologically in a priority queue. One idea was to manage all the events in one global event queue, but as there may be thousands of events that have to be stored, inserting new events in the queue would be rather inefficient. The chosen solution was to maintain an event queue for every node (with events concerning one node, such as get online, download a page, replace tunnel...) and every tunnel step (with events concerning the sending and receiving of data). This resulted in thousands of small queues instead of one huge storage. On the other side, a synchronization mechanism was needed to ensure that all events occurred in their correct chronological order. This problem was solved by dividing the simulated time into slots of fixed size (some milliseconds) and doing an iteration over all nodes (or their event queues respectively) in every slot. When the nodes, their links and tunnels where further marked with flags telling whether they are currently active (sending or receiving some data), the simulation could be speed up by a factor of at least ten! The design details are easiest explained with an example, which will be described during the next few chapters.

4 Initializing the Simulator

4.1 Specifying the Test Environment

The simulator is not designed to simulate just one specific network. On the contrary, the idea is that the user first specifies a test case scenario in which he describes the network that should be simulated. This is done in a properties file, the mechanism is further described in chapter 9. As it may be useful to have several different properties files at hand, the user does not have to overwrite the same properties file again and again but may manage several different properties files under different names. Therefore, the first thing to do when starting a simulation is to tell the Simulator from which properties file it should read the network settings. The first argument passed to the Simulator is therefore the name of the properties file to be used in the simulation (without its extension, see Appendix B for an example).
The second argument passed to the Simulator[3] is the period of real time that should be simulated (in milliseconds). The simulated time is not stored in the properties file as it will probably be the setting that is changed the most and its value is not really part of the network specification.

4.1.1 Initialize Properties Reader

The first thing done by the Simulator is to set a properties reader to the properties file passed as first argument[4]. If the file is found, the Simulator reads in the chosen amount of milliseconds simulated in one round[5]. In real time, the Nodes are acting simultaneously, so the smaller the period simulated in one round, the more precise the simulation. On the other side, reducing this value will slow down the simulator, as more rounds will be needed. With the second argument (the simulated real time) the Simulator may then calculate the needed number of rounds.

4.1.2 Initialize Simulator

When the PropertiesReader is successfully set, the Simulator is initialized. As there should never be more than one simulator running at the same time, the Simulator is implementing the Singleton pattern. For this, all initializations and accesses to the Simulator will be done with the method Simulator.getInstance(), which generates a new Simulator instance by calling its Constructor[6] when none is existing yet or returns the one and only already existing instance. As this is the first time the method is called, the method will now create and return a new Simulator instance by calling the Simulator constructor. The constructor will then initialize the Simulator. It first sets the variable counting the number of passed rounds in the simulation to zero. Afterwards, the number of simulated nodes is read in from the properties file[7] and an array is created to store all these nodes.

4.1.3 Initialize Logger

The Simulator can store events occurring during the simulation in a log file. For this it creates a new instance of the class Logger[8], which is reading in its logging properties from the file "logging.properties" and tries to create a new log file. The details of the Logger are further described in chapter 10.

4.1.4 Initialize Random Number Generator

The Simulator will produce events at random intervals with the help of a random number generator. The used generator is the class Random.java in the standard package java.util, which allows to create random Integer and Long values in a given interval[9].

[3]see method Simulator.main()
[4]see constructor PropertiesReader
[5]see property "SimulatedTimePerRound" in chapter 9.1.1
[6]see constructor Simulator
[7]see property "NumberOfNodes" in chapter 9.1.1
[8]see constructor Logger
[9]see class java.util.Random

4.2 Generate Network Environment

The Simulator knows now everything it needs to generate the test environment, which includes nodes, a web server, links between the nodes and tunnels over several nodes[10].

4.2.1 Generate Web Server

In a real network, the Nodes would send their web requests to different web servers. But the decision which web server is answering a web request or even a kind of load balancing between the different web servers should not influence the behavior of the network as the Simulator should return information about the behavior of the MorphMix nodes and not about the behavior of the web servers.
The first idea was to simply ignore the web servers. The last node that gets the web request would then just wait some time and then generate and return a web reply, so simulating the sending of the web request to and receiving its reply from the web server. But this solution misses the effect that the last node needs some bandwidth for the connection to the web server, which influences the available bandwidth for the other links.
The chosen solution solved this problem without increasing the complexity of the simulation too much by adding just one web server[11] with unlimited up- and downstream bandwidth[12] that represents all the web servers in the real network.
To distinguish the nodes for logging purposes, each node gets a unique number. For the web server, this is "-1", so allowing the node number generation to start normally at zero for the first "normal" node.
The simulator has a method to access its unique web server[13]. If there should ever be reasons to simulate more than one web server, this method would have to be adapted to choose one of the existing servers (with whatever criteria: load balancing, probabilities, at random) and all the web servers must be included in the simulators round loop over all nodes. The web servers could have the numbers -1,-2,-3... and the iteration over all nodes would include a special loop over all nodes with negative node numbers.

4.2.2 Generate Nodes

After the web server is generated, the Simulator creates all the normal nodes[14]. The simulated number of nodes has already be read in when the simulator got instantiated (see chapter 4.1.2 for details). Every node gets a unique node number. The vectors holding the nodes links and tunnels to other nodes get initiated, but remain empty. To increase the simulator's performance, every node holds separate sets with the actually sending and receiving links as well. The implementation of these two extra sets has increased the simulator's speed by a factor of at least 10! Every node maintains an event queue, which holds all events concerning this node sorted after its time of occurrence[15]. The node finally keeps account about all received and all failed requests, so there are two more collections to be initialized. At last, every node gets an up- and a downstream bandwidth[16] and an uptime pattern, which states over a 24-hour interval at which time a node is acting as a MIX node (is online) and during which period a node itself is sending or receiving data through the MorphMix system (is browsing). These three parameters (up- and downstream bandwidth, uptime pattern) are dependent on each other, as users with a good Internet access are probably longer online than users surfing with a slow modem. Users with low bandwidths will use their online time more carefully, as they will have to pay for the time they are active, where else user of unlimited internet access (ADSL, T1) will perhaps run the application for 24 hours a day. To consider these dependencies, the properties are all read in using the same random value.

[10] see method Simulator.init()
[11] see constructor WebServer
[12] see property "WebServerBandwidth" in chapter 9.1.1
[13] see method Simulator.getWebserver()
[14] see constructor Node
[15] see class EventPriorityQueue
[16] see properties "UpstreamBandwidth", "DownstreamBandwidth", "BandwidthUsage" in chapter 9.1.1

Generate Uptime Pattern The uptime pattern describes the nodes status over a period of 24 hours[17]. After 24 hours, the same uptime pattern starts again. The pattern describes the time a node is online (runs the MorphMix application) and the part of this period a node is actually browsing. As described above, this behavior depends on the nodes bandwidths (see chapter 4.2.2). To simplify the situation, every node is starting the application exactly once per day (or it is running it 24 hours without interruption). If a node has a small bandwidth (modem, ISDN), it is assumed that it is using all its online time for browsing. The time nodes with small bandwidth use for surfing is therefore identical to the time these nodes are online.
Fast nodes on the other side may be online for a long time (even 24 hours a day)[18]. They will therefore not use all their online time for browsing and every node gets a browsing interval that is part of their online period[19].
After the nodes online and browsing intervals are determined, the nodes status may be set (is online, is browsing). When all the parameters are set, the node is added to the simulator's collection of nodes. The status of the simulated network after the generation of the web server and all other nodes is illustrated in figure 3.

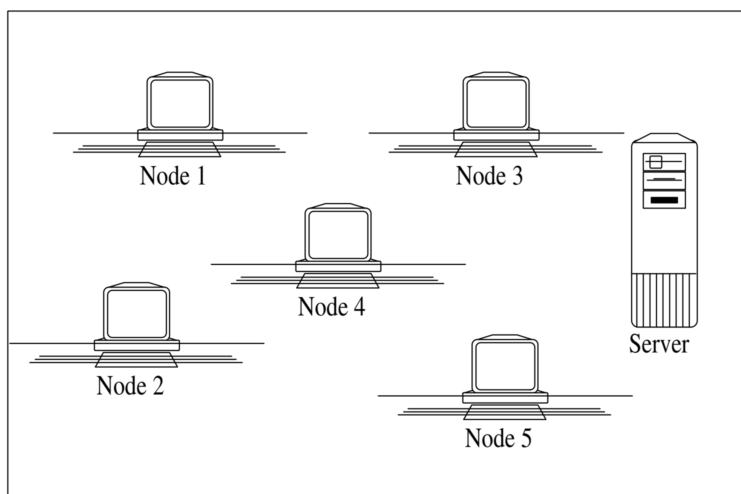

Figure 3: Unconnected Network with Server and Nodes

4.2.3 Generate Links to Web Server

As explained in chapter 4.2.1, the simulation maintains only one web server. As every node should be able to forward web requests to the web server, every node (except the web server itself) gets a link to the server when it is set online. A link is not directed, this means that it may be used by both nodes to send and receive data. This will be further explained when we come to TunnelSteps (see chapter 4.2.6). The link to the web server gets a special bandwidth and delay[20]. The link is then added to the collections of links from the node as well as to those of the web server[21]. After this is done for every node, the web server has a link to every other node and in return every node has a connection to the web server. The status of the simulated network after the generation of the links to the web server is illustrated in figure 4.

[17] see method Node.setUptimePattern()
[18] see property "ActiveInterval" in chapter 9.1.1 and method PropertiesReader.getActiveIntervalTime() for the calculation of its distribution
[19] see property "SendingTime"
[20] see properties "WebServerBandwidth" and "WebServerDelay" in chapter 9.1.1
[21] see method Node.addLink()

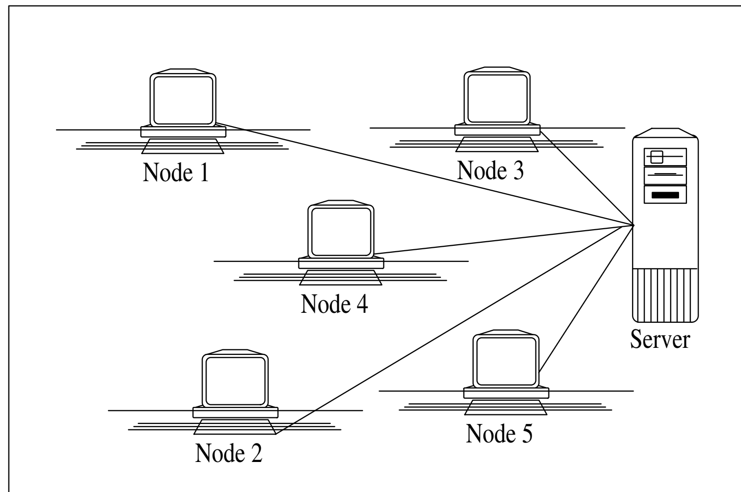

Figure 4: Network with Connections to the Server

4.2.4 Generate Links to other Nodes

Of course a node needs to know about some other nodes in the network as well, as it will have to choose one of them at random when generating a tunnel. As there is no solution for this task in the MorphMix protocol yet, every active node just gets a random set of other nodes[22]. The simulator first creates a set of random active nodes and guarantees that the updated node is not part of it. It checks whether the updated node already has a link to the random nodes[23]. If it doesn't have one, a new link is created with random bandwidth and delay[24]. The status of the simulated network when every node has initialized some links to other nodes is illustrated in figure 5.

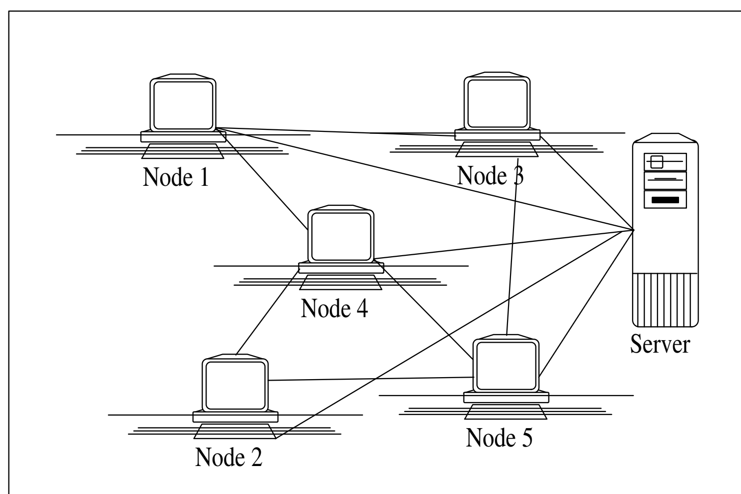

Figure 5: Network with Connections between Nodes

[22]see method Node.updateLinks() and property "NumberOfNodesInUpdate" in chapter 9.1.1
[23]see method Node.hasNodeToLink()
[24]see constructor Link and properties "LinkBandwidth", "LinkDelay" in chapter 9.1.1

4.2.5 Generate Tunnels through Network

Tunnels are the anonymous paths a message takes through the net. As it takes some time to establish such a connection over several nodes, every node holds a set of tunnels[25] ready for usage. To complicate traffic analysis attacks, tunnels are replaced after a while[26](e.g. 5 minutes). All tunnels have the same length[27](the number of MIXes a message has to pass before it is forwarded to the web server). If the property is set to n, the tunnel includes the initiator, n-1 random other nodes and the web server. Figure 6 illustrates such a tunnel. The path through the net is stored as an array of TunnelSteps, which represent the part of a tunnel between two nodes.

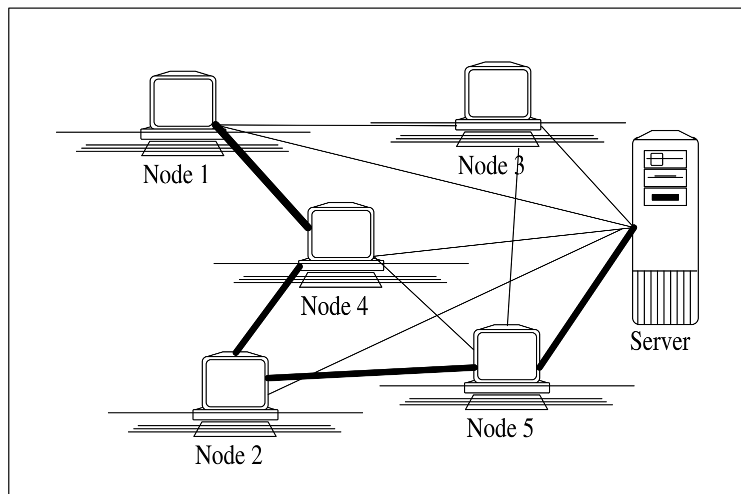

Figure 6: Tunnel from Node 1 to the Server

4.2.6 Generate TunnelSteps

To create a TunnelStep, we must know the sender for this step and choose one of its links at random[28]. If it is ensured that the tunnel doesn't include the newly chosen node already, the TunnelStep can be generated[29] and added to the Tunnel. The TunnelStep knows about the link it belongs to, the tunnel it is part of and about its sender and receiver. It has a send queue for packets it should forward[30] and an event queue with events telling it about the data it should receive[31]. It includes some flow control mechanisms to ensure that nodes aren't flooded with more data than they are able to handle (see chapter 7.3). If a TunnelStep isn't used for some time, its sender or receiver have probably left the network. It should then be removed. To find out when this has happened, every TunnelStep maintains a timeout counter, which decreases its value every round it is not used. If some data is sent over the tunnel, its timeout counter is reset to its initial value[32].
After the Tunnel and the Link are informed about their new TunnelStep[33], the Tunnel generation continues with the Tunnel's next step. The last node will not be chosen at random, as it must be the web server itself. The generated tunnel is then added to its initiator. As the message must return over the same way it has reached the web server, a similar tunnel is created from the

[25]see property "NumberOfTunnels" in chapter 9.1.1 and method Node.initTunnels()
[26]see property "ReplaceTunnel" in chapter 9.1.1
[27]see property "NumberOfHopsPerWay" in chapter 9.1.1
[28]see method Node.getRandomLink()
[29]see constructor TunnelStep
[30]see class java.util.LinkedList
[31]see class EventPriorityQueue
[32]see property "TunnelTimeout" in chapter 9.1.1
[33]see method Link.addTunnelStep()

web server to the initiator, linked with the other tunnel[34] and added to the web servers set of tunnels. Figure 7 shows a detailed view of a tunnel with all its tunnel steps.

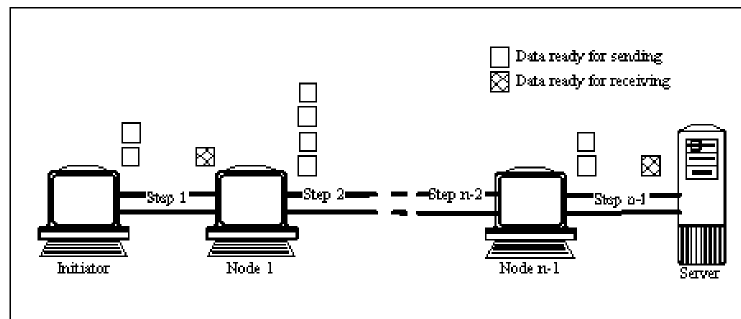

Figure 7: Tunnel from Node 1 to the Server in Detail

[34]see method Tunnel.setReturnTunnel()

5 Simulation: Iterating in Rounds over all Nodes

After the logger and the properties reader got initialized (see chapter 4.1.3 and 4.1.1) and the network with web server, nodes, links and tunnels got created (see chapter 4.2), the Simulator is ready to start to simulate network traffic. It divides the simulated time into slots of fixed length[35]. In every time slot, it iterates over all nodes and simulates their sending and receiving of data[36].

5.1 Simulate Web Server

The web server is treated separately as we must not check for several events that may influence the behavior of normal nodes (e.g. node activation or removal, starting of new web requests, tunnel replacements, timeouts). It is enough to check for data ready to be sent or received. These procedures are identical to the sending and receiving methods for normal nodes (see chapter 7 and 8).

5.2 Loop over all Nodes

The network is simulated by going in loops over all nodes and checking whether they have some events for this time slot waiting or data ready for sending or receiving.

5.2.1 Update Links

During the simulation, the set of links to other nodes every node holds gets smaller as nodes leave the network. It is therefore necessary to update the nodes' collection of links from time to time[37]. If the interval has passed, a process similar to the process of initializing the nodes' link set takes place (see chapter 4.2.4).

5.2.2 Set Node Offline

If the nodes' uptime pattern (see chapter 4.2.2) tells the node to leave the network in this time slot, the node is set offline[38]. The nodes' flags, telling whether it is currently online, active and browsing, are all set to false. There are two ways a node may leave the network: it either closes the MorphMix application correctly or crashes[39]. In case of a crash, all the nodes' links are removed immediately[40]. To remove a link, all the tunnels using this link have to be removed as well[41]. As this is a crash, the tunnels owner does not get informed about the interrupted tunnel. They will recognize the disconnected tunnel later when the tunnel will time out as it can no longer forward any data and a TunnelInteruptedTimeout is generated at the initiator (see chapter 5.2.5 for details). This means the initiator needs much longer to learn about the interrupted tunnel. In case of a shutdown, the initiators get informed immediately about the disconnected tunnel with a TunnelInterruptedEvent, which is only delayed some milliseconds simulating the time the message would need to reach the initiator[42].

5.2.3 Start Browsing

If the nodes uptime pattern (see chapter 4.2.2) tells the node to start browsing, the node will start a new web request[43]. This process is further described in chapter 6.

[35] see property "TimeslotSize" in chapter 9.1.1
[36] see method Simulator.run()
[37] see property "UpdateLinksInterval" in chapter 9.1.1
[38] see method Node.setOffline()
[39] see properties "Shutdown" and "Crash" in chapter 9.1.1
[40] see method Node.removeLink()
[41] see method Link.removeTunnelSteps()
[42] see property "InformRequestorDelay" in chapter 9.1.1
[43] see method Node.startBrowsing()

5.2.4 Stop Browsing

If the nodes uptime pattern (see chapter 4.2.2) tells the node to stop browsing, the nodes is browsing flag is set to false, so the node will not download any new pages[44].

5.2.5 Check for Node Events

All time-related events that may occur during a simulation are stored in the nodes event queue. It is therefore necessary that every active node checks its queue for events taking place in this round[45].

Start WebRequest Event A StartWebRequestEvent is added to the event queue when a node has downloaded a whole page. It reminds the node that it should start a new request after some delay simulating the user's time to read the page and clicking on the next link (see chapter 8.5.2 for details). If a StartWebRequestEvent takes place, the concerned node will start a new web request. This process is further described in chapter 6.

Replace Tunnel Event A ReplaceTunnelEvent is added to the event queue when a node has initialized a new tunnel (see chapter 4.2.5). It reminds the node after some time[46] that it should replace the tunnel. It the tunnel (and its corresponding return tunnel) is not transmitting any messages at the moments, all the pointers to this tunnel and its tunnel steps are removed from the belonging nodes and links and all its elements are set to null to assist the garbage collector. If there are still some messages using this tunnel, the tunnels replace flag is set, ensuring that no new messages are sent over this tunnel and that it is replaced as soon as the last message using this tunnel has reached its destination. In both cases a new SetupTunnelEvent is added to the nodes event queue, which reminds the node that the simulated replacement of a tunnel is finished and it may therefore add another tunnel to the nodes set of tunnels[47]. In the unlikely case that all tunnels should be replaced in the same interval, the last tunnel is used a bit longer until some other tunnels are created.

Setup Tunnel Event A SetupTunnelEvent is added to the event queue when a node has replaced one of its tunnels (see chapter 5.2.5). It reminds the node that the simulated replacement of this tunnel is finished and it may therefore add another tunnel to the nodes set of tunnels.

Tunnel Interrupted Event A TunnelInterruptedEvent is added to the nodes event queue when a node has left the network and the tunnel got interrupted (see chapter 5.2.2). The tunnel is then checked for unfinished requests and replies[48] and the node resends them over another tunnel.

5.2.6 Simulate Sending for Node

The process of sending some data in a time slot is described in chapter 7.

5.2.7 Simulate Receiving for Node

The process of receiving some data in a time slot is described in chapter 8.

5.2.8 Check Tunnels For Timeouts

As nodes enter and leave the network dynamically, tunnels may get interrupted. To find disconnected tunnels, the simulator remebmbers the time a tunnel was last used. If there are tunnels that should be transmitting data, but haven't neither sent or received any data for some time[49],

[44] see method Node.stopBrowsing()
[45] see method Node.checkForEvents()
[46] see property "TunnelTimeout" in chapter 9.1.1
[47] see property "TunnelSetupTime" in chapter 9.1.1
[48] see method TunnelStep.getUnfinishedMessages()
[49] see property "TunnelTimeout" in chapter 9.1.1

it is assumed that the tunnel got interrupted. To enable this checks, every tunnel has a timeout counter, which is decreased every round[50]. If the timeout gets zero, the tunnel seems to be interrupted somewhere and all the messages should be resent over another tunnel. This is done by adding a TunnelInteruptedEvent to the owner's event queue[51]. The time the information would need to reach the initiator is simulated by delaying the event for some time[52].

5.3 Print simulator status

After some rounds[53], the simulation prints a small status report.

```
...
Round 4601/6000. Estimated time left: 364 sec.
00:46.000 Status: Online: 1985, Active: 305, Browsing: 315.
Ongoing page downloads: 79, requests: 25, replies: 273.
...
```

A status report as illustrated above includes the following information:

- "Round 4601/6000" shows that the Simulator is simulating the 4601st of a total of 6000 rounds.
- "Estimated time left: 364 sec" gives an assumption about the time until the simulation is finished. This is just a simple forecast from the time the application needed to simulate the last few rounds.
- "00:46.000" states the simulated time. The application has simulated 46 seconds of real time so far.
- "Online: 1985" shows the number of nodes that are currently acting as a MIX node.
- "Active: 305" shows the number of nodes that are currently sending or receiving data (either their own downloads or messages they are forwarding for other nodes).
- "Browsing: 315" shows the number of nodes that are currently browsing the net. They may either be downloading or just reading a page.
- "Ongoing page downloads: 79" states the number of pages that are currently downloaded.
- "requests: 25" shows the number of requests that are on the way from their requestor to the web server.
- "requests: 273" shows the number of replies that are on their way from the web server back to their requestor.

An analysis of the returned states can be found in chapter 12.

[50] see method Node.checkTunnels() and Tunnel.decreaseTimeout()
[51] see method Tunnel.informRequestors()
[52] see property "InformRequestorDelay" in chapter 9.1.1
[53] see property "LogStatus" in chapter 9.1.1

6 Start Web Request

The simulator's primary task is to simulate web requests and replies. But the simulator design, especially the usage of test case scenarios enables to extend the simulator to other net traffic such as mail or FTP.

6.1 Generate Web Request Message

To simulate the request of a new web page, the initiating node creates a new Message of type "WebRequestIndex"[54], so it can be distinguished from web requests that are downloading objects embedded in an index file, which are called "WebRequest". The messages the web server returns are called "WebReplyIndex" and "WebReply". Every Message knows about its initiator and the tunnel it follows through the net. The messages are numerated so they can be distinguished easily. Messages embedded in index file "n" are called "n-1", "n-2" and so on. Every message gets a length[55] and one of the initiator's tunnels is chosen at random as the requests path through the network[56]. To find out how long a message needs to find its way to the web server and back, every message gets timestamped. Another timestamp is used to calculate the time needed to download a whole page, which not only includes the request for the page's index file, but the following request's for the objects embedded in the index file as well.

6.1.1 Generate Packets

The message gets divided into packets of fixed size[57]. This calculation has to take care of the bytes needed for the packet header. The last packet will probably not be filled to the last bit, but we do not want to send smaller packets as they could be traced easily. The packets know about the message they are part of and their actual sender and receiver[58], which at initialization time are the tunnels first and second node. As packets aren't forwarded to the next hop before they are totally received, no packet will ever be sent or received by more than one node at the same time. To find out when a packet is sent or received totally, every packet counts its already sent or received bytes.

6.2 Send Web Request

After the message and its packet are instantiated, the message can be sent over its allocated tunnel. This is done by adding all its packets to the first tunnel steps send queue[59]. The message itself is added to the tunnels set of running messages to simplify the analysis whether a tunnel is still in use. The node also counts the number of requests it has sent out.

[54] see method Node.startWebRequest()
[55] see property "WebRequestLength" in chapter 9.1.1
[56] see method Node.getRandomTunnel()
[57] see properties "PacketSize" and "PacketHeaderSize" in chapter 9.1.1 and Message constructor
[58] see method Message.createPackets() and Packet constructor
[59] see methods Tunnel.sendMessage() and TunnelStep.addPacketToSendQueue()

7 Simulate Sending

There are many nodes that are online, but neither receiving nor sending any data at the moment. To improve the simulation's performance, the nodes that are actually processing some data (sending or receiving) are marked with a "isActive" flag, which allows to ignore all the passive nodes. Every node has a fixed bandwidth which it may use to send data. But there may be several links with several tunnels that want to send some data. So the first thing to do is to distribute the nodes bandwidth between all the objects with data waiting to be sent.

7.1 Distribute Bandwidth among Links

The first distribution step is to share the bandwidth between all links with waiting data to be sent[60]. To find them, the simulator must loop over all tunnels for every link in its set of sending links and check if it has at least one packet waiting in its sending queue[61]. When the number of sending links is determined, the size of the bandwidth share for one link may be calculated and the send method for each link gets called. Figure 8 shows an example: The sending node has a bandwidth of 80 bytes for this round. It has four links, three of them have some data in their send queue. The 80 bytes are divided distributed among these three links and every link gets 26 Bytes which it may now further share among its tunnels. If a links' bandwidth is smaller than the share it got from the node, it may not use the full share. This is not a problem at all, as the unused bandwidth will be re-shared among all the other links (see chapter 7.5). As a link may have several tunnels with data ready for sending, the links bandwidth share has to be divided again among the sending tunnels[62].

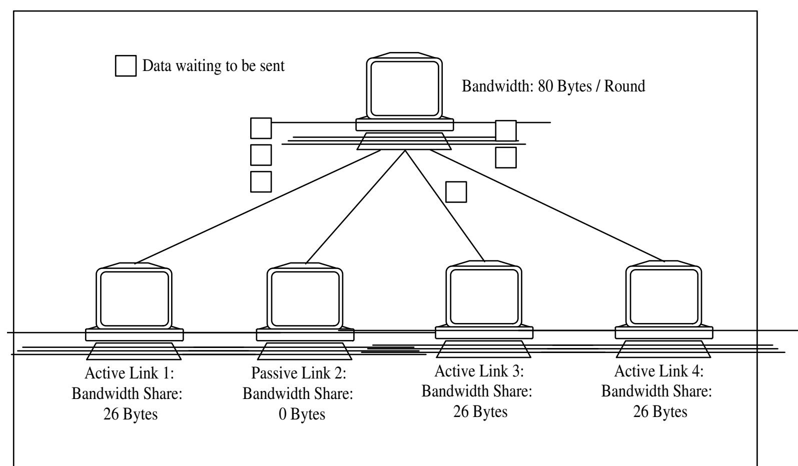

Figure 8: Bandwidth Sharing among active Links

7.2 Distribute Bandwidth among Tunnels

The links bandwidth share can now be further distributed among the sending tunnel steps. A check is needed to find all the tunnel steps with waiting packets in their send queue[63]. When the number of sending tunnels is determined, the size of the bandwidth share for one tunnel may be calculated and the send method for the tunnel gets called[64]. Figure 9 completes the example from figure 8. The sending node shares its bandwidth among its three sending links, so each of them gets a bandwidth share of 26 bytes per round. The link distributes this share again among its tunnels with data waiting to be sent. In the example, the two active tunnels each receive its share of 13 bytes per round which is now used for sending.

[60] see method Node.send()
[61] see method Node.getSendingLinks()
[62] see method Link.send()
[63] see method Link.getSendingTunnelSteps()
[64] see method TunnelStep.send()

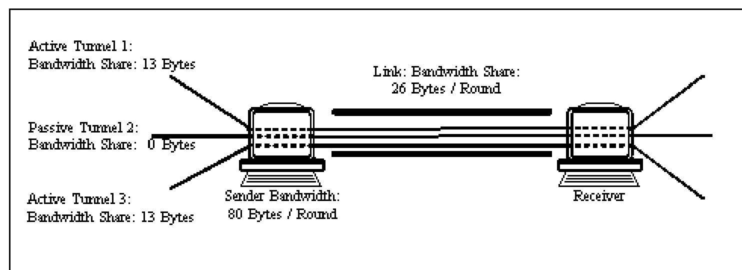

Figure 9: Bandwidth Sharing among active Tunnels

7.3 Flow Control

As the sender and receiver may have different bandwidth shares for this tunnel, a control mechanism is needed so that a fast sender does not flood the receiver with more data than it can possibly handle. There are two flow controls implemented in the simulator, a TCP window control simulation and a credit based flow control. These flow controls can not only be used to prevent slow receivers from getting flooded, but are quite useful to find interrupted tunnels as well.

7.3.1 TCP window flow control

Every tunnel has a buffer with an initial value of some thousand bytes[65]. Every time the sender tries to transfer some bytes, this buffer is decreased and when the node at the other end has received some bytes, the buffer gets increased again. If the buffer goes to zero, the sender may not send more data until the receiver has done some receiving again. This control is called TCP window control as a similar control mechanism is implemented in the TCP protocol. In contrary to the simulator, the control in TCP is implemented on the receiver side. The protocol guarantees that the number of bytes sent by the sender but not yet received by the receiver doesn't exceed the fixed limit. It is as well useful to find out whether the node on the other side is still active or not. If the tunnels bandwidth share is bigger than the buffer size, not all of the bandwidth may be used for sending. This is not a problem at all, as the unused bandwidth will be re-shared among all the other sending tunnels (see chapter 7.5).

7.3.2 Credit based flow control

Now its time for the second flow control. It restricts the number of packets a node may send over one tunnel [4]. When the tunnel gets initialized, its sender gets a fixed amount of packets it may send over it[66]. When the credit is used up, the sender must wait. But if the receiver has received a fixed amount of packets[67], it increases the senders packet credits again and the sending may continue. So the second flow control ensures that a slow receiver doesn't get flooded as well as the TCP window control, but on another level.

7.4 Simulate Sending of some Bytes

When both checks have been passed, the actual sending of the data may take place. The sending is simulated by increasing the packets sent bytes counter. It will return if the packet is full and some unused bandwidth is left. Now the buffer for the TCP window flow control gets decreased. Then the receiver at the other end of the tunnel has to get notified that the sender has simulated the transmission of some data and that it should simulate its receiving. This is done by adding a ReceiveBytesEvent to the receiver's event queue. It is delayed a bit, so simulating the tunnels delay[68].

[65] see property "TunnelBufferSize" in chapter 9.1.1
[66] see property "TunnelCredits" in chapter9.1.1
[67] see property "NeededPacketsToReturnCredits" in chapter 9.1.1
[68] see property "LinkDelay" in chapter 9.1.1

If the packet is completely sent and some bandwidth is left unused, it will either be used to send the next packet (or at least a part of it) or returned to the link so it can be shared among the other tunnels. The totally transmitted packet gets removed from the send queue and the sent packets counter gets increased[69]. If the whole message has been sent, the tunnel is no longer used for this message and can be set waiting.

7.5 Re-share unused Bandwidth

As mentioned above, there may be tunnels and links that do not need their whole bandwidth share. As this bandwidth should not just be lost, it may be re-shared among the other sending tunnels and links. There is a check that the returned shares are significant enough as the re-sharing is not worth the effort for a few bytes[70].

[69]see method Tunnel.sentPacket()()
[70]see properties "BandwidthResharingNode" and "BandwidthResharingLink" in chapter 9.1.1

8 Simulate Receiving

The process of receiving data is handled similar to the sending mechanism described in chapter 7. Every node has a fixed bandwidth which it may use ro receive data. But there may be several links with several tunnels that want to receive some data. So the first thing is to distribute the bandwidth between all the objects with data waiting to be received[71].

8.1 Distribute Bandwidth among Links

To find all the links with data waiting to be received the simulator must loop over all tunnels for every link in the nodes set of receiving links and check if it has at least one event waiting in its sending queue[72]. It must be ensured that the next event in the queue takes place in this time slot (or earlier). When the number of sending links is determined, the size of a bandwidth share for one link may be calculated and the receive method for the links gets called[73]. If the links bandwidth is smaller than the share it got from the node, it may not use the full share. This is not a problem at all, as the unused bandwidth will be re-shared among all the other links (see chapter 8.6). As a link may have several tunnels with data ready for receiving, the links bandwidth share has to be divided again among the receiving tunnels.

8.2 Distribute Bandwidth among Tunnels

The link's bandwidth share can now be further distributed among the tunnel steps with packets in their send queue[74]. When the number of receiving tunnels is determined, the size of the bandwidth share for one link may be calculated and the receive method for the tunnel gets called[75].

8.3 Simulate receiving of some Bytes

The receiving of data is handled with events. This events have a time of occurrence, so when getting a new ReceiveBytesEvent from the event queue, it must be ensured that its data has already arrived at the receiver by checking its time of occurrence. To make this easier, the events in the queue are ordered chronologically. So if the first event has not taken place yet, checking the other events is useless, as they occur even later. The tunnel step must always have a packet ready in which it can write the received bytes. If it doesn't have one yet, a new packet is opened[76]. While there are actual events in the queue and the tunnel's bandwidth share is not used up yet, events are processed. The receiving of the data is simulated by increasing the packets received bytes counter. We can be sure that an event does never hold data of more than one packet as this is the way the creation of events is handled (see chapter 7.4). After the data is received, the TCP window buffer must be increased again (a description of this flow control can be found in chapter 7.3.1). When the whole event is received, it gets removed from the event queue and the next event gets processed. If the tunnels bandwidth share is almost used up and doesn't allow to receive the whole event, the events size is decreased, but the event remains in the event queue, so it can be processed again in the next round. If not all the bandwidth is used to receive data, it is returned to the link and re-shared among the other tunnels (see chapter 8.6).

8.4 Receive a whole Packet

If a whole packet is received[77], the received packet counter gets increased and eventually some credits are returned to the sender[78] (see description of credit based flow control mechanism in

[71] see method Node.receive()
[72] see method Node.getReceivingLinks()
[73] see method Link.receive()
[74] see method Link.getReceivingTunnelSteps()
[75] see method TunnelStep.receive()
[76] see method TunnelStep.receivePacket()
[77] see method TunnelStep.receivedPacket()
[78] see property "NeededPacketsToReturnCredit" in chapter 9.1.1

chapter 7.3.2). The receiver must now decide what should happen to the packet.

Packet reached Web Server If the receiver is the web server, the packet has reached its destination and nothing has to be done as the web server will wait until it has received the whole message and then generate a reply.

Packet reached Requestor If the packet is part of a web reply and has reached its requestor, nothing has to be done, as the receiver will wait until it has received the whole message.

Forward Packet In all other cases the packet is added to the next tunnel steps send queue. The packets sender and receiver are adjusted and its bytes counter are reset to zero[79]. If the message has more packets, the next packet is received[80]. Otherwise, the whole message has reached the node.

8.5 Receive a whole Message

The receiving of the whole message[81] only has consequences if the receiver is the web server or the requestor of a received web reply.

8.5.1 Message reached Web Server

When the web server receives a web request, it will generate a reply. The received message is removed from the tunnels set of running messages[82]. If the tunnel doesn't have any more running messages, it is no longer used and may be removed. This is only the case if the tunnels remove flag is set (since it was used long enough and should be replaced, see chapter 5.2.5). The requestor is informed that its request has reached the server so it can decrease the counter for the number of ongoing requests and increase the number of ongoing replies. The generated reply is a message of type "WebReply", its number is the same as the one of the web request. Its size is calculated with the Pareto distribution[83] stated in (1), which will mostly return a value between 1 and 10 KB, but may generate really huge values as well.

$$Filesize(Bytes) = \frac{2.4}{(1 - random[0,1])^{\frac{1}{1.2}}} - 2.4 \quad (1)$$

It is returned in the same way it has reached the web server but using the tunnel's return tunnel. Now that the way is set the message's packets may be generated and added to the first tunnel step's send queue. The request message is no longer used and may be removed (this may help the Java Virtual Machine's Garbage Collector).

8.5.2 Message reached Requestor

The requesting node is informed that it has received a reply[84]. If the reply is the index file of a web page (message is of type "WebReplyIndex"), the receiver will now request all the objects included in the web page and therefore generate zero or more additional web requests. These will be of type "WebRequest" to distinguish them from requests for the index files, which are called "WebRequestIndex". The number of objects in an index file is calculated with the Pareto distribution[85] stated in (2).

$$\#EmbeddedObjects = \frac{0.8}{(1 - random[0,1])^{\frac{1}{1.2}}} - 0.8 \quad (2)$$

[79] see methods Tunnel.getNextTunnelStep(), Packet.setSender(), Packet.setReceiver(), Packet.initBytes()
[80] see method Message.getActuallyReceivedPacket()
[81] see method TunnelStep.receivedMessage()
[82] see method Tunnel.removeRunningMessage()
[83] see method PropertiesReader.getFileSizePareto()
[84] see method Node.receivedReply()
[85] see method PropertiesReader.getNumberOfObjects()

Depending on the simulated HTTP version[86] , the following requests are either requested in parallel over different tunnels (HTTP1.0) or sequentially over the same tunnel(HTTP 1.1). If the reply is one of the objects in a already received index file, it is checked whether the whole page is received. In this case, a StartWebRequestEvent[87] is added to the nodes event queue. It has a delay to simulate the passed time between the receiving of a web page and the time the user clicks on the next link. This delay is calculated with a negative exponential distribution (see (3) and an average browse delay taken from the properties file[88].

$$BrowseDelay(ms) = \frac{-\ln(1 - random[0,1])}{\frac{\ln 0.5}{Delay_{average}}} \quad (3)$$

The request message is no longer used and gets removed (this may help the Java Virtual Machine's Garbage Collector).

8.6 Re-share unused Bandwidth

As mentioned above, there may be tunnels and links that do not need their whole bandwidth share. As this bandwidth should not just be lost, it may be re-shared among the other receiving tunnels and links. There should be a check that the returned shares are significant enough as the re-sharing is not worth the effort for some few bytes[89].

[86]see property "HTTPVersion" in chapter 9.1.1
[87]see class StartWebRequestEvent
[88]see PropertiesReader.getBrowseDelay() and property "AverageBrowseDelay" in chapter9.1.1
[89]see properties "BandwidthResharingNode" and "BandwidthResharingLink" in chapter 9.1.1

9 Using properties files to set the test environment

The simulator has several parameters that depend on the environment the user wants to simulate, e.g. the number of nodes the network consists of. There is a standard solution to handle all these parameters in Java applications called properties files. The application reads all the parameters out of a text file. The user therefore doesn't have to search all the parameters in the code of the different classes.
The MorphMix simulator will use two properties files; the first holds all the settings for the logger and will be explained in the chapter about the functionality of the logging mechanism (see chapter 10). The second properties file describes all the settings for the simulated network.

9.1 The Test Case Properties File "WebRequest.properties"

All the parameters specifying the simulated network are stored in one properties file called "WebRequest.properties". Since testers will probably like to test the behavior of the MorphMix simulator under different circumstances, it may be useful to work with several different properties file with different names.

9.1.1 The properties used for setting the simulated network environment

All values concerning periods of time are stored in milliseconds, data amounts in bytes and probabilities in percentage.

TimeslotSize This value defines how many milliseconds are simulated in one round. The smaller the period, the exacter, but also the slower the simulation. A good value would be 10 ms.

Pareto The sizes of web files (index files and its embedded objects) and the number of objects embedded in an index file can be approximated with a Pareto distribution. If the property "Pareto" is set to "no" (instead of "yes"), the distribution is replaced by a simpler distribution with minimal and maximal values (see chapter 9.2.2 for details).

FileSize The size of a file returned ba the web server (either an index file or an object embedded in an index file) is approximated with the Pareto distribution (see chapter 9.2.5 for details). It will produce a lot of small values of a few kilo bytes, but is not limited with a maximal size. So in very rare cases, there may be files with a size of several gigabytes. As this may influence the simulation too much, it is useful to limit the maximal file size produced by the Pareto distribution. A good limit would be 50 megabytes.

NumberOfObjects When a node has received an index file from the web server, it requests all the objects embedded in it in separate requests (if there are any). The number of embedded objects is approximated with the Pareto distribution (see chapter 9.2.4 for details). It will produce a lot of small values, but is not limited with a maximal size. So in very rare cases, there may be several thousand objects embedded in a n index file. As this may influence the simulator too extremely, it is useful to limit the maximal number of embedded objects produced by the Pareto distribution. A good limit would be 100 embedded objects.

HTTPProtocol There are two http protocols that may be simulated by the application, HTTP1.0 and HTTP1.1. To only difference between the two protocols that concerns the behavior of the simulator is their way of requesting the objects embedded in an index file. HTTP1.0 is requesting over simultaneously over different TCP connections (therefore using different MorphMix tunnels), where HTTP1.1 is using the same TCP connection for all embedded objects that it has already used for the index file.

NumberOfTunnels The instantiation of a new tunnel needs some time. It is therefore useful that every node maintains several already instantiated tunnels ready for usage.

PacketSize The messages transferred over the network are divided into packets. Their size must be fixed to complicate traffic analysis.

PacketHeaderSize The messages transferred over the network are divided into packets. As in other Internet protocols, the packet holds some information about its size, source and destination. This information needs some bytes which can not be used for the data to be transferred.

NumberOfHopsPerWay The number of nodes a message passes before it is forwarded to the web server. This number does include the initiator at the beginning, but not the web server at the end of the tunnel.
So if "NumberOfHopsPerWay" = 3, a tunnel would look as follows:

Node1(Initiator) -> Node2 -> Node3 -> Server

NumberOfNodesInUpdate Every node has to know about some other nodes in the network if it wants to establish tunnels. The way the node is informed about other nodes in the network is not yet specified. The simulator is therefore just updating all nodes with a set of other active nodes. The maximal number of nodes included in such an update is stored in the property "NumberOfNodesInUpdate".

UpdateLinksInterval Every node has to know about some other nodes in the network if it wants to establish tunnels. The way the node is informed about other nodes in the network is not yet specified. The simulator is therefore just updating all nodes with a set of other active nodes. The period of time between two updates is stored in the property "UpdateLinksInterval".

TunnelBufferSize The simulator includes several flow control mechanisms to ensure that a receiver is not flooded with more data than it is able to handle. One of this controls is the TCP window protocol. It uses a buffer in which all the data that is sent out but not yet received is stored temporarily. If the buffer is full, the receiver is not able to keep up with which the sender and the sender has to wait until there is some free space in the buffer again.

TunnelCredits The simulator includes several flow control mechanisms to ensure that a receiver is not flooded with more data than it is able to handle. One of this controls is based on credits. Every sender gets some initial credits, which it uses to send packets. If some packets are received, the receiver returns their credits to the sender. If the sender has no credits left, the receiver is not able to keep up with which the sender and the sender has to wait until it gets some credits returned.

NeededPacketsToReturnCredit The simulator includes several flow control mechanisms to ensure that a receiver is not flooded with more data than it is able to handle. One of this controls is based on credits. Every sender gets some initial credits, which it uses to send packets. If the receiver has received the number of packets specified in the property "NeededPacketsToReturnCredit", it returns these credits to the sender. If the sender has no credits left, the receiver is not able to hold the speed with which the sender is transferring data and the sender has to wait until it gets some credits returned. The used value is often half the size of the number of tunnel credits specified in the property "TunnelCredits".

ReplaceTunnel Every node maintains some tunnels over which it may send requests. These tunnels are replaced after some time to complicate traffic analysis. The period a tunnel is used is stored in the property "ReplaceTunnel".

BandwidthResharingNode The bandwidth one node has to send data is split up between all sending (or receiving) links. If a link does not need its full bandwidth, it is returned to the node and reshared among the other links if the returned bandwidths size is worth the effort.

BandwidthResharingLink The bandwidth one node has to send data is split up between all sending (or receiving) links. The link shares its part of the nodes bandwidth among all its sending (or receiving) tunnels. If a tunnel does not need its full bandwidth, it is returned to the link and reshared among the other tunnels if the returned bandwidths size is worth the effort.

NumberOfNodes The size of a simulation depends on the simulated period of time and the number of simulated nodes.

WebRequestLength The length of a request sent to the web server doesn't vary as much as the size of the replies.

UpstreamBandwidth The amount of data a node may send per round is limited by its up-stream bandwidth.

Downstreambandwidth The amount of data a node may receive per round is limited by its downstream bandwidth.

BandwidthUsage There may be nodes running other Internet applications than MorphMix. They will not be interested that all their bandwidth is used up for forwarding other nodes messages. The property "BandwidthUsage" states how many percents of its bandwidth (upstream and downstream) a node is offering for the MorphMix application.

LinkBandwidth The amount of data a link may transfer per round is limited by its bandwidth.

LinkDelay The time some data needs to be sent over a link and reach the next node depends on the links delay.

WebServerBandwidth The simulator is only simulating one web server with almost unlimited bandwidth.

WebServerDelay The time the web server needs to generate a reply depends on its delay.

ActiveInterval Every node has an uptime pattern which states the time a node is online or browsing (over 24 hours). The period of time a node is active is stated in percentages of a full day.

AverageBrowseDelay The time in milliseconds a user needs to read a page and click on the next link or enter the next URL.

BrowsingTimeSuperUser The time a user with unlimited Internet access is browsing per day. This time doesn't have to be specified for users with a modem or an ISDN as it is assumed that they will use all their online time for browsing.

TunnelSetupTime The time in milliseconds needed to instantiate a new tunnel. During this time the node will only have a limited number of tunnels it may choose to send requests.

InformRequestorDelay If a node is shutting down the application correctly, it will inform all the nodes that are using tunnels over the leaving node that they must replace their tunnels and resend not yet returned requests over another tunnel. The time in milliseconds until this information has reached the nodes is stored in this property "InformRequestorDelay".

TunnelTimeout If a node is crashing, the nodes that are using tunnels over it do not get informed that they must replace their tunnels and resend not yet returned requests over another tunnel. The data sent over this tunnel will therefore just get stuck somewhere. After some time this is recognized by the nodes that are not able to send or receive data from this node and they will inform the owner of the tunnel. But this takes a lot longer than a correct shutdown.

Shutdown A node leaving the network is either shutting down the application correctly or crashing. In the first case, it will inform all the nodes that are using tunnels over it that they must replace their tunnels and resend not yet returned requests over another tunnel. The probability of a shutdown is stored in the property "Shutdown".

Crash A node leaving the network is either shutting down the application correctly or crashing. In case of a crash, the nodes that are using tunnels over it do not get informed that they must replace their tunnels and resend not yet returned requests over another tunnel. The probability of a crash is stored in the property "Crash".

LogStatus After some rounds specified in this value, some informations about the simulations current status are printed out and logged (see chapter 5.3 at page 18 for details).

9.2 Distributions

As some of the property values are not always the same, the class "PropertiesReader" can be used to access the values stored in the properties files in different ways.

9.2.1 Fixed Value

The normal solution known from other properties files is to have a property that has always the same value. Figure **??** shows the setting for the property with key "NumberOfNodes". The returned value will always be 10000.

Properties File

```
..
NumberOfNodes=10000
..
```

Figure 10: Number of nodes included in the network with fixed value

9.2.2 Random Distributed Property with Minimal and Maximal Value

This properties have a minimal and a maximal value specified in the properties file. The two values must have the name of the property and the extension "Min" and "Max". The properties reader will then return a random value somewhere between the two extremes. Figure 9.2.2 shows the setting for the property with key "LinkDelay". The returned value will be a random value equally distributed between 10 and 20.

Properties File

```
..
LinkDelayMin=10
LinkDelayMax=20
..
```

Figure 11: Link Delay with Min and Max

9.2.3 Property with several possible Values and their Probabilities

The properties file holds several possible values that might be returned and their probabilities. The returned value will be one of the specified possibilities. The probabilities of all possible values should add up to 100%. Figure 9.2.3 shows the setting for the property with key "LinkBandwidth". The returned value will be 56 with 60% probability, 128 with 30% probability and 256 with 10% probability.

```
Properties File
..
LinkBandwidth1=56
LinkBandwidthProbability1=60
LinkBandwidth2=128
LinkBandwidthProbability=30
LinkBandwidth3=256
LinkProbability=10
..
```

Figure 12: Link Bandwidth with three possible values and their probabilities

9.2.4 Pareto Distribution for Number of Embedded Objects in an Index File

The number of objects embedded in an index file is approximated with the Pareto distribution (see formula (1)). The formula does not need any inputs, the only properties specified in the properties file concerning the Pareto distribution are therefore the value "MaxNumberOfEmbeddedObjects" to prevent some huge index files to distort the simulation.

9.2.5 Pareto Distribution for File Size

The file size of an index file or an embedded object is approximated with the Pareto distribution (see formula (2) at page 24). The formula does not need any inputs, the only properties specified in the properties file concerning the Pareto distribution are therefore the value "MaxFileSize" to prevent some unrealistic number of embedded objects to distort the simulation.

9.2.6 Browse Delay

The time a used needs to read a downloaded page and until he starts to download the next page is calculated with a negative exponential distribution (see formula (3)).

9.3 The logging properties file

The logging properties file holds all settings concerning the behavior of the logging mechanism. A description of the logging mechanism can be found in chapter 10.

10 The logging mechanism

To analyze the behavior of the simulated network, the simulator should log occurring events. Furthermore, there must be tools ready to analyze the log files because the log files can become quite long. The simulator is designed to be able to generate log files for different purposes. When debugging or testing one small functionality of the simulator, useful log files should state every smallest event, e.g. the sending of some bytes to the next node or the creation of a new tunnel. On the other side, logging all these events in a big test with several 1000 rounds and nodes would generate unmanageably long log files and significantly slow down the performance of the simulator too much. To make the tool adaptable to different logging purposes, the logger knows seven different logging depths. When the simulator is started, the logger's depth is set to the property value "LoggingDepth" specified in the logging properties file called "Logging.properties". The concerned entry in the logging properties file will look like this:

LoggingDepth=3

All events sent to the logger for logging are only logged if their logging state is at least as high as the logger's depth. The logging depth of the different events, e.g. SendBytes, SendPacket, SendMessage, StartWebRequest, are specified in the logging properties file as well. The logging events are divided into the following logging classes:

- Depth1: Errors. Something went wrong.
- Depth2: Warning. Something was not correct, but could be fixed.
- Depth3: The most important events, such as the starting of a web request and the return of a web reply.
- Depth4: Events of depth "message", messages getting sent and reaching a node.
- Depth5: Events of depth "packet", packets getting sent and reaching a node.
- Depth6: Events of depth "Bytes", some bytes getting sent and received.
- Depth7: Changes of the environment, such as nodes getting active, added links or tunnels.

To adapt the logger for a special purpose, the user has two possibilities: he either changes the logger's logging depth by changing the property "LoggingDepth" in the logging properties file or he changes the logging depths of the different events. When analyzing the behavior of one node, the logging depth should be set to 4 or deeper, as higher depths will not show what data one special node is sending and receiving. To analyze the functionality of the implemented flow controls (i.e. bandwidth splitting), the depth must be set to 6. On the other side, analyzing when a node is active and when it is passive is easiest done by increasing the logging depths of the events "SetActiveNodePassive" and "SetPassiveNodeActive". The log files are written in a simple, semi colon separated text format that allows easy parsing and analyzing by machine(see chapter 11 for an example).

11 Analysis

The simulator includes several tools to analyze the generated log files. This includes a tool analyzing the behavior of the nodes over 24 hours[90], one to analyze the behavior of the nodes during the simulation[91], one to analyze the messages sent over the net[92] and last but not least one to analyze messages that have not reached their destination[93]. Another tool can be used to find all logging messages with a special string[94]. Of course this can be done with tools like grep as well. The simulator is running these tools automatically after the simulation has finished. The success of the analysis tools can depend on the simulators logging status; if nothing gets logged, nothing can be analyzed. Some examples about produced statistics can be found in chapter 12.

11.1 Status Analysis

The class "StatusAnalyzer" produces statistics about the average number of nodes that are online (running MorphMix), active (sending or receiving data), browsing or downloading pages. It calculates the average number of messages on their way to the web server, the average number of replies on their way back to their requestor and the average number of page downloads.

11.2 Uptime Pattern Analysis

The class "UptimePatternAnalyzer" produces statistics about the number of online and browsing nodes over 24 hours. It calculates the average time a node is online or browsing as well.

11.3 Message Analysis

The class "MessageAnalyzer" produces statistics about the total number of downloaded pages, requests and replies and their total and average size. It analyzes the average and median time a request needed to reach the server, the time a request needed until the requestor gets its first byte, until it gets the whole request and until it gets the whole page. Large files with many embedded objects will be downloaded much slower because several objects have be requested sequentially over the same tunnel. It is therefore useful to analyze the times of the index files for themselves, as they are not influenced by sequential downloads. Last but not least, the analyzed times are stored in several files.

11.4 Failure Analyzer

The class "FailureAnalyzer" produces statistics about the messages that did not reach their destination. It counts the number of used tunnels that got disconnected because a node is leaving the network and the number of messages that had to be resent because of these interrupted tunnels.

[90] see class UptimePatternAnalyzer
[91] see class StatusAnalyzer
[92] see class MessageAnalyzer
[93] see class FailureAnalyzer
[94] see class StringAnalyzer

12 Testing

The simulation uses different functions that can be tested for correctness. This includes the various distribution functions in the class PropertiesReader. It is important to check the results generated by the simulator. As the MorphMix protocol does not exist yet, it is only possible to check the simulated data for internal consistency.

12.1 Analyzing one Request

To understand the behavior of the sending and receiving algorithms, it may be interesting to look at one request in detail. This can be done by setting the property "LoggingStatus" in the file "logging.properties" to 6. This ensures that everything gets logged. As the simulation may generate thousands of logging messages, it is best to stop the simulation after the first pages have been downloaded (when the number of ongoing page downloads gets decreased). Find all the messages for one particular page download. If we are looking at the first started page download, we would first search for a message with the string "WebRequestIndex 1" included. There are several possibilities to search for occurrences of a string. The analysis package includes the class Simulator.analysis.StringAnalyzer, which can be used to search for strings. The passed arguments should be the name of the log file and the searched string (if the string consists of more than one word as in "WebRequestIndex 1", it is important to include the words in quotation marks", as it will otherwise search only for the first word). Of course standard search tools can be used as well. If we start at the top of the log file, we would find a line similar to this one:

```
...
00:00.000;SendPacket;P0(WebRequestIndex 1,N40,N3194,T40-1)
...
```

Now we know that the message was sent over tunnel T40-1. If we search the whole file for messages with the string "T40-1" included, we would find all messages concerning this tunnel. This could look the following:

```
...
00:00.000;Tunnel;T40-1(N40, N3194, N2066, N3950, N3554, N-1)
00:00.000;Tunnel;T40-1(N-1, N3554, N3950, N2066, N3194, N40)
...
```

These first two lines log the initialisation of the tunnel. As every tunnel needs a return tunnel (see 4.2.5 for details), two tunnels are actually instantiated. These lines allow to check that the tunnels have the requested length from the property "NumberOfHopsPerWay". In the test case, this property is set to 5, which means that every tunnel should include the initiator, 4 random nodes and finally the web server. Next it is possible to compare the two tunnels and guarantee that the return tunnel does go over the same nodes as the other tunnel (but in the other direction).
After the instantiation, the tunnel can be used for sending:

```
...
00:00.000;Request;WebRequestIndex1(208,T40-1)
...
```

The first line logs the start of request number 1. The first number in parenthesis is the requests size (208 bytes), the second the tunnel it is sent over.

```
...
00:00.000;SendPacket;P0(WebRequestIndex 1,N40,N3194,T40-1)
...
```

In the same round, node N40 starts to send the messages first (and only) packet. It will be forwarded to node N3194 over tunnel T40-1.

```
...
00:00.000;SendBytes;80(WebRequestIndex 1,N40,N3194,T40-1)
00:00.010;SendBytes;80(WebRequestIndex 1,N40,N3194,T40-1)
...
```

Still in the first round, the node n40 sends the first 80 bytes of request 1 to node n3194. This allows to check if the nodes bandwidth per round is calculated correctly. As the property "TimeSimulatedInRound" is set to 10, the node is able to send 80 bytes per round or 8000 bytes per second. The node's bandwidth can be found by searching for "N40". The first returned line will be the node initialisation log message:

```
...
00:00.000;Node;N40(BW:80/80,O:85152651-3072651,S:85152651-3072651)
...
```

The line shows that the node has an upstream bandwidth of 80 bytes per round and a downstream bandwidth of 80 bytes per round as well. The other numbers are the node's online and sending interval. Node n40 will continue to send more data during the next rounds.

```
...
00:00.010;ReceiveBytes;80(WebRequestIndex 1,N40,N3194,T40-1)
00:00.020;SendBytes;80(WebRequestIndex 1,N40,N3194,T40-1)
00:00.020;ReceiveBytes;80(WebRequestIndex 1,N40,N3194,T40-1)
00:00.030;SendBytes;80(WebRequestIndex 1,N40,N3194,T40-1)
...
```

10 ms after node n40 has sent out the first bytes, node n3194 receives them. This tells us that node n3194 has a bandwidth of at least 8000 bytes as well. A check with the initialisation log message:

```
...
00:00.000;Node;N3194(BW:320/320,A:78399471-78399471,S:79779458-579458)
...
```

The delay between node n40 and node n3194 is 10 ms. This is correct as it is the value specified in the property "LinkDelay" as well. Now it would be possible to count the number of sent and received bytes. If 1000 bytes have been sent, a new log message will appear:

```
...
00:00.120;SendBytes;40(WebRequestIndex 1,N40,N3194,T40-1)
00:00.120;SentPacket;P0(WebRequestIndex 1,N40,N3194,T40-1)
00:00.120;SentMessage;WebRequestIndex 1(N40,N3194,T40-1)
...
```

After 120 ms, node n40 is sending the last bytes of the first packet. As node n40 can send 80 bytes per second, it will need 13 rounds to send the whole packet (1000 bytes). As the first round starts at 0:00.000, the 13st round will be at 00:00.120. It logs that it has finished sending the packet. As the message only consists of one packet, the message is sent completely as well. Node n40 may now lean back and wait for the reply, while node n3194 is still receiving the request's last bytes.

```
...
00:00.130;ReceiveBytes;40(WebRequestIndex 1,N40,N3194,T40-1)
00:00.130;ReceivedPacket;P0(WebRequestIndex 1,N40,N3194,T40-1)
00:00.130;ReceivedMessage;WebRequestIndex 1(N40,N3194,T40-1)
00:00.140;SendPacket;P0(WebRequestIndex 1,N3194,N2066,T40-1)
00:00.140;SendBytes;320(WebRequestIndex 1,N3194,N2066,T40-1)
...
```

10 ms later, node n3194 receives the last bytes of the request. It will now start to forward the request to the tunnel's next node, which is node n2066. It seems that node n3194 has a bigger bandwidth, which is approved by the log file:

```
...
00:00.000;Node;N3194(BW:320/320,A:78399471-78399471,S:79779458-579458)
...
```

After some more time, the node finally reaches the web server:

```
...
00:00.390;ReceivedMessage;WebRequestIndex 1(N3554,N-1,T40-1)
00:00.390;MessageReachedServer;WebRequestIndex 1(208,T40-1,390)
00:00.390;Reply;WebReplyIndex 1(7811,N40,T40-1)
00:00.400;SendPacket;P0(WebReplyIndex 1,N-1,N3554,T40-1)
...
```

The needed time was 390 ms, which is logged as the last number in the "MessageReached-Server" message. If all nodes would have to work with minimal bandwidth, a time of 5 * 120ms + 5 * 10ms (delay) = 650 ms would have been needed. As there are faster nodes included in the tunnel, the 390 ms seem to be quite realistic. The server is now creating a reply of 7811 bytes and sends it back over node n3554.

```
...
00:01.660;ReceivedMessage;WebReplyIndex 1(N3194,N40,T40-1)
00:01.660;MessageReachedRequestor;WebReplyIndex1(7811,N-1,N40,T40-
1,1660)
...
```

After 1660 ms the reply reaches its requestor. The needed time is logged as the last number in the message "MessageReachedRequestor". Now the same procedure will start for the embedded objects, which can be checked as well by searching for the strings "WebRequest1-".

```
...
00:01.660;Request;WebRequest1-0(437,T40-3)
00:01.660;Request;WebRequest1-1(237,T40-2)
00:01.660;Request;WebRequest1-2(353,T40-3)
00:01.660;Request;WebRequest1-3(367,T40-1)
...
```

If all the embedded objects are received, the download has completed and the receiving of the whole page will be logged:

```
...
00:02.720;MessageReachedRequestor;WebReply1-3(258,N-1,N40,T40-1,1060)
00:04.700;MessageReachedRequestor;WebReply1-0(1790,N-1,N40,T40-3,3040)
00:05.550;MessageReachedRequestor;WebReply1-2(2599,N-1,N40,T40-3,3890)
00:05.850;MessageReachedRequestor;WebReply1-1(2060,N-1,N40,T40-2,4190)
00:05.850;ReceivedPage;N40(14518,4,5850)
...
```

As a final check, the pages size and time can be inspected. The time to download the whole page is 5850 ms. The pages size is 14518 bytes, which can as well be calculated from the size of the index file (7811 bytes) and its embedded objects(258 + 1790 + 2599 + 2060 bytes).
Node n40 will now need some time to read the downloaded page and will then start to download the next page.

12.2 Correctness Distributions

The correctness of the various distributions used to set up a randomized network environment can be tested with a special test class[95]. It calculates several thousand random parameter settings, which allows to check that the calculated values lie in the expected area and are distributed correctly.

12.3 Plausibility tests for number of online, active, browsing, downloading nodes

A nodes activity over 24 hours is stored in its uptime pattern. As a simple approach to simulate the behavior of network users, all nodes use their MorphMix application exactly once per day (see chapter 4.2.2 for details). There's a analysis tool[96] included in the simulation package which allows to check that this uptime pattern is plausible, e.g. that there are more or less the same number of nodes active all the time (as their time of activity is randomly distributed over 24 hours, so simulating users from all global time zones).
Depending on the parameter settings in the property file, the expected number of active nodes can be calculated and compared with the results of the uptime pattern analysis. The status and uptime pattern analysis tools return the following results:

	Test1	Test2	Test3	Average	Expected	Difference
Average online nodes	1984	1915	1937	1945	1912	1.31%
Average active nodes	386	388	439	404	-	-
Average browsing nodes	315	320	310	315	333	4.9%
Average downloading nodes	106	109	121	112	-	-
Average active time(s)	33539	33685	32769	33331	33048	0.86%
Average browsing time(s)	5439	5540	5550	5509	5760	4.36%

Table 1: Results Node Analysis

Calculations:
Average number of active nodes:

$$\begin{aligned} Nodes_{online} &= (p_{max} * Max + p_{min} * Min + p_{1-p_{max}-p_{min}} * \frac{Max + Min}{2}) * 5000 \\ &= (0.2 * 100 + 0.5 * 0.05 + (1 - 0.2 - 0.5) * \frac{1 + 0.05}{2}) * 5000 \\ &= 1912.5 \end{aligned} \quad (4)$$

Average number of browsing nodes:

$$\begin{aligned} Nodes_{surfing} &= (p_{min} * Min * p_{MinBrowsing} + (1 - p_{min}) * Max * p_{MaxBrowsing}) * 5000 \\ &= (0.5 * 0.05 + (1 - 0.5) * 1.0 * \frac{2}{24}) * 5000 \\ &= 333 \end{aligned} \quad (5)$$

Average time a node is active:

$$\begin{aligned} Time_{active} &= (p_{max} * Max + p_{min} * Min + (1 - p_{min} - p_{max}) * \frac{Max + Min}{2}) * 100\% \\ &= (0.2 * 1.0 + 0.5 * 0.05 + (1 - 0.5 - 0.2) * \frac{1.0 + 0.05}{2} * 100\% \\ &= 38.25\% (33048s) \end{aligned} \quad (6)$$

[95]see class TestDistributions
[96]see class UptimePatternAnalyzer

Average time a node is browsing:

$$
\begin{aligned}
Time_{surfing} &= ((p_{min} * Min * p_{MinBrowsing} + (1 - p_{min}) * Max * p_{MaxBrowsing}) * 100\% \quad (7)\\
&= (0.5 * 0.05 + (1 - 0.5) * 1.0 * \frac{2}{24}) * 100\% \\
&= 6.67\% (5760s)
\end{aligned}
$$

With the settings from the properties file above:

pmax:	Percentage of nodes with maximal activity (20%).
pmax:	Percentage of nodes with maximal activity (20%).
pmin:	Percentage of nodes with minimal activity (50%).
Max:	Percentage of time a node with maximal activity is online (100%).
Min:	Percentage of time a node with minimal activity is online (5%).
MaxSurfing:	Percentage of its active time a node with maximal activity is browsing.
MinSurfing:	Percentage of its active time a node with minimal activity is browsing.
	All other nodes have equally distributed periods of activity between Min and Max.

The differences between the expected values and the test results are all in a range of +/-5%. As the tests weren't done over a very long period, this range is acceptable.
Expected values for the number of active and downloading nodes aren't that easy to calculate. But the statistics return an average time of 5 seconds. As the average period of time a user needs to read a downloaded page is set to 10 seconds (see property "AverageBrowseDelay"), a node spends one third of its online time for downloading and two thirds for reading. This is exactly the ratio browsing to downloading nodes.
Every request is forwarded over 4 other node, so every request needs at any time minimally two nodes and maximally 5 nodes that forward its data. The ratio active to downloading nodes is 3.6, so a message is averagely distributed over 3.6 nodes.

12.4 Plausibility tests for needed Time

The class MessageAnalyzer calculates the average and median time a message needs to reach the server, the time until the first byte of the message has returned to its requestor and the time needed to download the whole file (either an index file or an embedded object) and the whole page. Large files with many embedded objects will be downloaded much slower, as several objects have be requested sequentially over the same tunnel. It is therefore useful to analyze the times of the index files for themselves, as they are not influenced by sequential downloads. Some small tests return the following values:

The average number of ongoing page downloads is of course identical to the average number of downloading nodes (see 12.3). The ratio page downloads to requests is 1007 to 3625. As a page download needs 5324 ms and a request 1662 ms, the expected average number of ongoing requests can be approximated with the following formula:

$$
\begin{aligned}
\#OngoingRequests &= \#OngoingPageLoads * \frac{Time_{page}}{Time_{request}} * \frac{\#Pages}{\#Requests} \quad (8)\\
&= 112 * \frac{1662}{5324} * \frac{3625}{1007} \\
&= 126.86
\end{aligned}
$$

The ratio page downloads to replies is 1007 to 3271. As a page download needs 5324 ms and

	Test1	Test2	Test3	Average
Average ongoing page downloads	106	109	121	112
Average ongoing requests	110	95	97	101
Average ongoing replies	313	316	327	319
Total page downloads	976	1072	973	1007
Total requests	3452	3712	3710	3625
Total replies	3052	3329	3433	3271
Average time request index file(ms)	733	707	721	720
Average time reply first byte index file	2583	2535	2734	2617
Average time reply index file(ms)	2732	2693	2983	2803
Average time request(ms)	1832	1537	1617	1662
Average time reply first byte(ms)	2583	2535	2734	2617
Average time reply:(ms)	5571	5031	5701	5434
Average time page load(ms)	5205	4980	5788	5324
Median time request index file(ms)	690	670	690	683
Median time reply index file(ms)	1920	1770	1850	1847
Median time page load(ms)	3470	3070	3440	3327

Table 2: Results Time Analysis

a reply 5434 ms, the expected average number of ongoing replies can be approximated with the following formula:

$$\begin{aligned} \#OngoingReplies &= \#OngoingPageLoads * \frac{Time_{reply}}{Time_{page}} * \frac{\#Replies}{\#Pages} \\ &= 112 * \frac{5434}{5434} * \frac{3271}{1007} \\ &= 371.32 \end{aligned} \quad (9)$$

12.5 Plausibility tests for sizes

The class MessageAnalyzer calculates the average and median sizes of requests, replies and whole pages. It also returns the average number of objects embedded in an index file. Some small tests return the following values:

	Test1	Test2	Test3	Average	Expected	Difference
Total page downloads	976	1072	973	1007	-	-
Total requests	3452	3712	3710	3625	-	-
Total replies	3052	3329	3433	3271	-	-
Total number of kb	18024	19040	20383	19149	-	-
Total number of kb for requests	1386	1492	1470	1449	-	-
Total number of kb for replies	16638	17548	18913	17700	-	-
Average size request(bytes)	401	401	396	399	400	0.25%
Average size reply(bytes)	5451	5271	5509	5410	5340	1.29%
Average page size(bytes)	17046	16369	19437	17617	17312	1.73%
Median size reply index file	1828	1984	1912	1908	-	-
Median size page(bytes)	4904	4513	4722	4713	-	-
Average # embedded objects	2.59	1.95	2.07	2.20	1.78	19.1%

Table 3: Results Size Analysis

The total number of bytes needed for requests should be identical to the product of the total number of requests and the average size of a request (and similarly for replies and page

downloads):

$$\begin{aligned} \#BytesRequests &= \#Requests * AverageSizeRequest \\ &= 3625 * 399 \\ &= 1446375 \end{aligned} \tag{10}$$

$$\begin{aligned} \#BytesReplies &= \#Replies * AverageSizeReplies \\ &= 3271 * 5410 \\ &= 17696110 \end{aligned} \tag{11}$$

$$\begin{aligned} \#Bytes &= \#PageLoads * AverageSizePage \\ &= 1007 * 17617 \\ &= 17740319 \end{aligned} \tag{12}$$

All calculated results correspond with the test values.
The average size of a request can be easily calculated as well, as it uses equally distributed values between a min(property "WebRequestLengthMin" set to 200 bytes) and a max(property "WebRequestLengthMay" set to 600 bytes):

$$\begin{aligned} RequestSize &= \frac{min + max}{2} \\ &= \frac{200 + 600}{2} \\ &= 400 \end{aligned} \tag{13}$$

The expected result for the average reply size can be calculated with an integral over the Pareto distribution (see formula 1).

$$\begin{aligned} FileSize(kb) &= \int_0^1 \frac{2.4}{(1-x)^{\frac{1}{1.2}}} - 2.4\,dx \\ &= 12 \end{aligned} \tag{14}$$

The result doesn't match with the value from table 3. But the Pareto distribution can produce extremely huge results. If we ignore the file sizes with the biggest results (as it is similarly done in the simulation), we got a result near the one from the table.

$$\begin{aligned} FileSize(kb) &= \int_0^{0.99} \frac{2.4}{(1-x)^{\frac{1}{1.2}}} - 2.4\,dx \\ &= 5.34 \end{aligned} \tag{15}$$

The Pareto distribution is used as well to calculate the number of objects embedded in an index file (see formula 2).

$$\begin{aligned} \#EmbeddedObjects &= \int_0^1 \frac{0.8}{(1-x)^{\frac{1}{1.2}}} - 0.8\,dx \\ &= 4 \end{aligned} \tag{16}$$

We ignore again the percent with the highest results.

$$\begin{aligned} \#EmbeddedObjects &= \int_0^{0.99} \frac{0.8}{(1-x)^{\frac{1}{1.2}}} - 0.8\,dx \\ &= 1.78 \end{aligned} \tag{17}$$

The result from the simulation lies somewhere between these two values. Another possibility to check the returned number of embedded objects is to calculate them from the number of page downloads and the number of replies (don't forget to subtract the index file):

$$\begin{aligned} \#EmbeddedObjects &= \frac{\#Replies}{\#PageLoads} - 1 \\ &= \frac{3271}{1007} - 1 \\ &= 2.25 \end{aligned} \tag{18}$$

The average page size can finally be calculated by multiplying the average reply size with the average number of embedded objects per page (don't forget to include the index file as well):

$$\begin{aligned} \#PageSize &= (\#EmbeddedObjects + 1) * ReplySize \\ &= (2.20 + 1) * 5410 \\ &= 17312 \end{aligned} \tag{19}$$

Except for the number of embedded objects, the differences between the expected values and the test results are all in a range of +/-5%. As the tests weren't done over a very long period, this range is acceptable.

13 Conclusions

The simulators actual status allows to run simulations on different network environments. It fulfills all the goals from the semester thesis. After some performance redesigns, the simulator is able to simulate the requested 10'000 nodes in real time or even faster, which is much more than could be hoped for at the beginning of the semester thesis. The code is optimized, but should still be readable for interested persons with some background in object oriented programming. It is documented richly with Javadoc and this semester thesis. The design of the simulator should be open enough to allow the implementation of further details. Interesting extensions are the improvement of the uptime pattern generation and the design of malicious nodes to test the protocols vulnerability towards different attacks. The simulator has been tested again and again during the whole development phase. The used properties files should enable testers to use the simulator for their own network environments without having to change any code at all.
Last but not least the implemented logging mechanism and the various analyzing tools should allow testers to analyze their test results easily.
The tests included in this document concentrate on plausibility checks to ensure the correct behavior of the code. The simulator is now in a stable and well documented state, which should allow to test its behavior for different environments and get ideas for the design of the MorphMix protocol.

14 Acknowledgements

I would like to thank Professor Bernhard Plattner and especially my supervisor Marc Rennhard for offering me such a great student thesis. He always found the time to discuss new aspects and problems and gave me a lot of thoughtful advice regarding the design of the simulator. I learnt a lot about network protocols in general and anonymous peer-to-peer networks in particular. The thesis also offered me the possibility to improve my experience in object-oriented design and consolidate my Java skills. I finally made my first experiences with LaTeX, which may be quite helpful when writing my diploma thesis. I hope you enjoyed reading this report as much as I enjoyed writing it.

Pascal Stucki, 28.02.2003

A Timetable

Week	1	2	3	4	5	6	7	8	9	10	11	12	13
read Documentation	xx												
Study other simulators design	x	xx	xx										
Create design		x	xx	xx	xx	x	x						
implement core functionality				x	xx	xx	xx	x					
Improve functionality							x	xx	xx	xx	x		
Testing								x	x	x	xx	x	
Report structuring				xx	xx	x	x						
Write report				x	xx	xx	x	x	x	x	xx	xx	xx

Table 4: Timetable

B How to use the MorphMix Simulator

B.1 Requirements

B.1.1 Simulator

You should have gotten the Simulator with all its Java source code and documentation in a ZIP-file called Simulator.zip.

B.1.2 Java Standard Development Kit

The Simulator is written in Java. It should therefore run under all common operation systems and was tested with Windows2000 and Solaris and Debian Linux. To compile and run the Simulator, a Java development kit is needed. It can be downloaded from

http://java.sun.com/j2se/1.4/download.html

There are versions for all kind of operation systems. The tests were done with Java1.4.

B.1.3 Zip Tool

The Simulator with all Java source code and documentation is packaged into a ZIP-file. A Tool such as "WinZip" for Windows and "unzip" for UNIX/LINUX is needed to unpack all the files from the ZIP-file.

B.2 Installing the Simulator

B.2.1 Unzipping the file Simulator.zip

The first step is to unzip the file Simulator.zip to a folder of your choice. Under Windows, you will probably use WinZip, under UNIX/Linux, most systems will already have the package unzip or something similar installed, which allows to unzip ZIP-files as well. All files will be unpacked in a directory called "MorphMixSimulator". It will include three subfolders:

- Subfolder "Simulator" holds all the Java files with the source code.
- Subfolder "Documentation" holds a file "MorphMixSimulatorHowTo.txt" with this manual and a subfolder "MorphMixSimulator" with the Javadoc description of the Java files.
- Subfolder "LoGFiles" will hold all log files generated when running the Simulator.

B.3 Compile the Java Files

The Java files are stored in the subfolder "Simulator" and its subdirectories "analyse" and "util". All java files in these three folders have to be compiled to generate class files. These can be done with the following command:

```
javac -server Simulator/*.java Simulator/util/*.java Simulator/analyse/*.java
```

B.4 Generate a Test Environment

The Simulator is getting all the settings about the network it should simulate from a properties file. So before starting the Simulator, a properties file needs to be created. You can find an example of such a properties file called "WebRequest.properties" in the directory "Source/PropertiesFiles". This file sets all the properties in a way to simulate a network with passed web requests and replies. When running the Simulator for the first time, you may just use this properties file or create your own by copying it and doing some small changes. The mechanism of the properties files are described in the semester thesis under chapter 9. There's a second properties file in the directory called "logging.properties". It holds information telling

the logger which events occurring in the simulation should be logged. The mechanism of the Logger is further described in the semester thesis under chapter 10. When you have adapted the properties files to your expectations, you may finally start the simulation.

B.5 Start the Simulator

There are two arguments that must be passed to the Simulator. The first is the name of the properties file generated under B.4, the second the simulated real time in milliseconds. Be careful: As the Simulator may have to simulate several thousands of Nodes, it may need up to one hour of processor time to simulate ten seconds Example:

Assume we have stored our network settings in the file "Webrequest.properties" and want to simulate one minute of real time network traffic. The simulator would then be started as follows:

java -server Simulator/Simulator WebRequest 60000

B.6 Analyze the Log File

The Simulator will generate a log file in the subfolder "LogFiles" with name "logfile+Date.log" where

- dd is the day on which the simulation took place.
- MM is the month in which the simulation took place.
- yyyy is the year in which the simulation took place.
- hh is the hour in which the simulation started.
- mm is the minute in which the simulation started.

For long situations with a lot of notes, the log file may be huge (several MB). It is therefore useful to use some log file analyzing tools to get the most important facts from the log file rather than reading the log file itself. There are several analyzing tools included in the distribution. See the chapters 11 and 12 for details).

References

[1] David L. Chaum, *Untraceable Electronic Mail, Return Adresses and Digital Pseudonyms*; Communications of the ACM, 24(2):84-88, February 1981.

[2] Marc Rennhard, *Peer-to-Peer based Anonymous Internet Usage with Collusion Detection*; TIK Technical Report Nr. 147, TIK, ETHZ Zurich, Zurich, CH, August 2002.

[3] M. Rennhard and B. Plattner, *Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection*;
Workshop on Privacy in the Electronic Society (WPES), in association with 9th ACM Conference on Computer and Communications Security (CCS 2002), Washington, DC, USA, November 21st 2002.

[4] M. Rennhard, S. Rafaeli, L. Mathy, B. Plattner, and D. Hutchison, *Analysis of an Anonymity Network for Web Browsing*;
In Proceedings of the IEEE 11th Intl. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2002), pages 49-54, Pittsburgh, USA, June 10th-12th 2002.