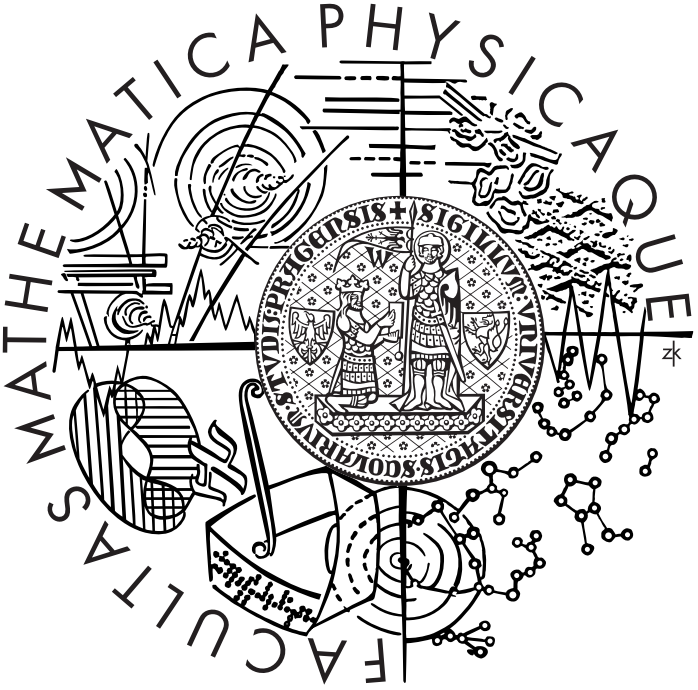
Charles University, Prague
Faculty of Mathematics and Physics

Master Thesis



Petr Matějka

Security in Peer-to-Peer Networks

Department of Software Engineering
Supervisor: Ing. Petr Tůma, Dr.
Study Program: Computer Science

I would like to thank Bruno Crispo and Bogdan Popescu for the guidance and insight they gave me during my time in Amsterdam. Petr Tůma, my supervisor in Prague, deserves acknowledgement for helpful suggestions and comments.

Without the efforts of Josh Guilfoyle (author of giFT), Turtle would never grow to a user friendly application. I am also very grateful to Rolan Yang for providing the Orkut data.

Special thanks go to Petra for her continuing support and patience.

I declare that I wrote the thesis by myself and listed all used sources. I agree with making the thesis publicly available.

Prague, December 5, 2004 Petr Matějka

Contents

Název práce: Security in Peer-to-Peer Networks
Autor: Petr Matějka
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Ing. Petr Tůma, Dr.
e-mail vedoucího: petr.tuma@mff.cuni.cz
Abstrakt: Tato diplomová práce popisuje peer-to-peer architekturu pro bezpečné sdílení citlivých dat *Turtle*. Hlavním přínosem nové architektury je způsob, jakým se staví k bezpečnostním otázkám: zatímco existující peer-to-peer architektury s podobným zaměřením budují důvěrné vazby nad základní, ne příliš důvěryhodnou síťovou infrastrukturou, Turtle využívá již předem existujících vztahů mezi svými uživateli k tomu, aby byla zajištěna anonymita jak odesílatele, tak příjemce dat. Zároveň jsou chráněny všechny mezilehlé uzly na cestě od odesílatele k příjemci dat.

Diplomová práce popisuje softwarovou architekturu Turtle, včetně detailů komunikačního protokolu. Jsou zde prezentovány výsledky výkonnostních testů, které potvrzují, že implementace netrpí významnými výkonnostními problémy. Následuje analýza grafových dat z Orkutu (web pro online komunitu) a práce je zakončena diskuzí o problémech, které ještě bude potřeba vyřešit.
Klíčová slova: peer-to-peer, sítě, anonymita, sdílení dat

Title: Security in Peer-to-Peer Networks
Author: Petr Matějka
Department: Department of Software Engineering
Supervisor: Ing. Petr Tůma, Dr.
Supervisor's e-mail address: petr.tuma@mff.cuni.cz
Abstract: In this thesis we present *Turtle*, a peer-to-peer architecture for safe sharing of sensitive data. The main contribution of Turtle rests in its novel way of dealing with trust issues: while existing peer-to-peer architectures with similar aims attempt to build trust relationships on top of the basic, trust-agnostic, peer-to-peer overlay, Turtle takes the opposite approach, and builds its overlay on top of pre-existent trust relationships among its users. This allows both data sender and receiver anonymity, while also protecting all intermediate relays in the data transfer path.

In the thesis, we will describe software architecture of Turtle, including details of communication protocol. Results of performance measurements are presented, verifying that our implementation of Turtle does not suffer from performance problems. Analysis of graph data from online community web site Orkut follows and we close with a list of problems that need to be solved.
Keywords: peer-to-peer, networks, anonymity, data sharing

Chapter 1

Introduction

Freedom to exchange information derives from the freedom of speech; unfortunately, there are many countries where this basic human right is not guaranteed. Turtle is a peer-to-peer data sharing architecture that makes very hard to restrict the freedom to exchange information by either technical or legal means.

The design of Turtle is inspired by the way how people living under oppressive regimes share information deemed hostile by their government. Because of the potentially very serious consequences raising from being caught possessing or distributing such material, no single individual is willing to share it, except with close friends. Experience has repeatedly shown that, even in the most repressive environments, this friends-to-friends delivery network is remarkably effective in disseminating information, with relatively little risks for the participating parties; if one chooses his friends carefully, the chance of being caught doing the forbidden exchanges becomes very small.

The idea behind Turtle is to take this friend-to-friend exchange to the digital world, and come up with a peer-to-peer architecture allowing private and secure sharing of sensitive information between a large number of users, over an untrusted network, in the absence of a central trust infrastructure.

There are three main contributions of this thesis. First, the model of Turtle described in [26] has been extended by adding support for virtual circuits, which makes sharing long files possible. Second, Turtle has been implemented and a variety of performance measurements were performed on the implementation. The implementation includes many advanced features known from other peer-to-peer file sharing applications, e.g. multisource downloading, hash-links, bandwidth management or metadata querying. And the third contribution is analysis of data from online community web site Orkut[25], which verifies one of the assumptions made in [26].

The rest of this thesis is organized as follows. In Chapter 2 we discuss

related work on peer-to-peer technologies. Chapter 3 explains model of Turtle network. Software architecture of Turtle node and communication protocol are described in Chapter 4. Performance measurements including discussion of the results are presented in Chapter 5. In Chapter 6 we analyze data from Orkut web site. Chapter 7 concludes the thesis and gives overview of future work.

Chapter 2

Other peer-to-peer networks

Peer-to-peer technologies have recently gained extreme popularity among researchers and users. There are many peer-to-peer networks already running and even more peer-to-peer protocols proposed. In this chapter we will summarize peer-to-peer technologies that are somewhat similar to Turtle.

2.1 Networks without anonymity

Peer-to-peer networks without anonymity do not try to hide identity of network users. They are usually faster than networks with anonymity and that is why they became popular especially for file sharing, where speed of search and download is an important property. However, all these networks are running into problems, because they are widely used for illegal sharing of copyrighted digital products. Although most of the networks cannot be simply shut down, their users are sued for providing copyrighted material to other users. Finding such user is as simple as joining the network and running a query. The network reveals IP addresses of users that answered the query, which is sufficient to identify the users.

Napster [24] In 1999 the first massive file sharing network called Napster was started. The network was not truly decentralized – it had centralized search servers that maintained index of all files available on the network. Because of this feature, it was possible to shut down the network due to violation of copyright laws.

FastTrack [22] After the Napster was shut down, FastTrack network with its Kazaa client became the number one file sharing network. It does not have any centralized servers – all nodes that join the network may

become search servers if they have enough capacity (memory, bandwidth, etc.).

Gnutella [20] Gnutella is another fully decentralized file sharing network. Its communication protocol is not as good as FastTrack protocol, but is fully documented. We will describe it in Section 2.1.1.

EDonkey2000 [11] The eDonkey file sharing network is decentralized peer-to-peer network with two kinds of nodes – servers and clients. Clients allow users to connect to the network and to share files. Servers act as meeting hubs for the clients. Because the server network is changing very often, there is a need for list of active servers. Such lists can be found on various web pages.

2.1.1 Gnutella[1]

Gnutella is a decentralized peer-to-peer file sharing protocol developed in 2000 by Nullsoft. Gnutella development was halted shortly after its results were made public, and the actual protocol was reverse engineered. Today there are numerous applications that employ the Gnutella protocol in their own individual way. The Gnutella Development Forum[9] was founded to merge different branches of protocol into single standard.

Gnutella nodes (*servents*) perform tasks normally associated with both servers and clients. They provide client-side interfaces through which users can issue queries and view search results, they accept queries from other servents, check for matches against their local files, and respond with corresponding results. Servents are also responsible for managing the background traffic needed to maintain network integrity.

In order to join the system a new node needs to know the IP address and port of any servent that is already connected. There are several known hosts that are almost always available and that can be initially connected. Once attached to the network (having one or more open connections with nodes already in the network), nodes send messages to interact with each other. Messages can be broadcasted (i.e., sent to all nodes with which the sender has open TCP connections) or back-propagated (i.e., sent on the reverse path as the initial broadcasted message). Several features of the protocol facilitate this broadcast/back-propagation mechanism:

1. Each message in the network has a unique identifier.

[1]This section is based on [28].

2. Nodes keep a short list of recently routed messages, which is used to prevent re-broadcasting and implement back-propagation.

3. Messages contain time-to-live counter that is decreased at every node on the path. When the counter reaches zeto, message is dropped.

There are many kinds of messages in Gnutella network, however we will concentrate only on a basic subset. The first two messages are *network membership messages*: PING and PONG. A node joining the network announces its presence by broadcasting PING message. Other nodes of the network reply with PONG messages, which are back-propagated to the new node. The PONG message contains information about the node such as its IP address and the number and size of shared files. In the dynamic environment where Gnutella operates, nodes often join and leave and network connections are unreliable. To cope with this environment a node periodically PINGs its neighbours to discover other participating nodes. Using information from received PONG messages, a disconnected node can always reconnect to the network.

File search is implemented using *search messages*: QUERY and QUERY HIT. QUERY message is broadcasted and contains a user specified search string that each receiving node matches against locally stored file names. QUERY HIT is back-propagated reply to QUERY message, which includes information necessary to download a file (IP address, file name, etc.).

The downloading process is done using the HTTP protocol based on the information extracted from the QUERY HIT message. Gnutella also supports file transfers from nodes that are behind firewall and cannot be contacted from outside world. Instead of downloading file from such node, PUSH message must be sent to the node. The node then initiates HTTP file upload (which is usually possible even on firewalled nodes).

Many extensions of Gnutella protocol have been proposed and implemented. The most important modification addresses the problem of network scalability. Nodes are now divided into two groups: *leaf nodes* and *ultrapeers*. Leaf nodes operate in the same way as described above. Ultrapeers run additional algorithms that reduce the need for broadcasting PING and QUERY messages everywhere. Because node discovery and file searching is done in a more efficient way, the networks is less loaded and is more scalable. The decision which nodes become ultrapeers is done by the network itself. It is dependent on the node capacity and on the current state of the network.

2.2 Networks with anonymity

There are many technologies that provide anonymity to their users or at least some of the users. These technologies bring anonymity into various services: e-mail, web browsing, file sharing or file storing. None of them is as widely deployed as non-anonymous file sharing peer-to-peer networks.

Anonymizer[2] Anonymizer is a http proxy that provides anonymity by proxying requests for web content on the user's behalf. It provides no protection for producers of information and does not protect consumers against logs kept by service itself.

Mixmaster[7] Mixmaster is an anonymous remailer used to deliver untraceable e-mails. It is based on Chaum's mix network design[5]. Chaum proposed hiding the correspondence between sender and recipient by wrapping messages in layers of public-key cryptography, and relaying them through a path composed of *mixes*. Each mix in turn decrypts, delays, and re-orders messages, before relaying them onward. Mix networks are resistant against traffic analysis, however, if a corrupt mix receives traffic from a non-core node, the mix can identify that node as the ultimate origin of the traffic.

Tor[30] Tor implements and extends Onion Routing schema originally proposed in [21]. Onion Routing is a distributed overlay network designed to anonymize low-latency TCP-based applications such as web browsing, secure shell, and instant messaging. Clients choose a path through the network and build a *circuit*, in which each node (or *onion router* or *OR*) in the path knows its predecessor and successor, but no other nodes in the circuit. The design is very similar to mix networks and suffers from same problem as described in previous paragraph.

Freenet[16, 6] Freenet is a self-organizing peer-to-peer network that provides file storage service. It focuses on providing anonymity for information producers, holders and consumers. See Section 2.2.1 for detailed description.

Tarzan[15] Tarzan is a peer-to-peer anonymous IP network overlay. Tarzan achieves its anonymity with layered encryption and multihop routing, much like mix networks. A message initiator chooses a path of peers pseudo-randomly through a restricted topology in a way that adversaries cannot easily influence. Cover traffic prevents a global observer from using traffic analysis to identify an initiator.

2.2.1 Freenet[2]

Each Freenet user runs a node that provides the network some storage space. To add a new file, a user sends the network an insert message containing the file and its location-independent globally unique identifier (GUID), which causes the file to be stored on some set of nodes. During file's lifetime, it might migrate to or be replicated on other nodes. To retrieve a file, a user sends a request message containing the GUID. Request is routed using steepest-ascent hill-climbing search (unlike Gnutella, which uses broadcasts). Each node forwards request to the node that it thinks is closest to the target. When the request reaches one of the nodes where the file is stored, that node passes the data back to the request's initiator.

Every Freenet node maintains a routing table that contains addresses of other nodes and the GUIDs it thinks they hold. When a node receives a query, it first checks its own store, and if it finds the file, returns it with a tag identifying itself as the data holder. Otherwise, the node forwards the request to the node in its table with the closest GUID to the one requested. That node then checks its store, and so on. If the request is successful, each node in the chain passes the file back and creates a new entry in its routing table associating the data holder with the requested GUID. Depending on its distance from the holder, each node might also cache a copy locally. To conceal the identity of the data holder, nodes occasionally modify reply messages, setting the holder tags to point to themselves before passing them back. Later requests will still locate the data because the node retains the true data holder's identity in its own routing table and forwards queries to the correct holder.

A subsequent query for the same GUID will tend to converge with the first request's path. Locally cached copy can satisfy the query after that happens. Subsequent queries for similar GUIDs will also jump over nodes to one that has previously supplied similar data. Nodes that reliably answer queries will be added to more routing tables, and will be contacted more often than nodes that do not.

To limit resource usage, queries contain a time-to-live counter that is decremented at each node. If the TTL expires, the query fails. If a cycle is detected during query propagation, the message is back-propagated and the node tries to use the next closest GUID instead of the previously used one. If a node runs out of candidates to try, it reports failure back to its predecessor on the path, which then tries its second choice, and so on.

Inserting files works in similar way as querying. An insert message follows the same path that a request for the same key would take, sets the routing

[2]This section is based on [6].

table entries in the same way, and stores the file on the same nodes.

Freenet GUIDs are calculated using SHA-1 secure hashes. The network employs three main types of keys. The content-hash key (CHK) is the low-level data-storage key and is generated by hashing the contents of the file to be stored. The keyword-signed key (KSK) is derived from a short descriptive chosen by the user when storing a file in the network. KSKs form a flat global namespace, which has several problems, e.g. they do not prevent inserting junk files under popular descriptions. The signed-subspace key (SSK) sets up a personal namespace that anyone can read but only its owner can write to. A user that owns the namespace publishes descriptive strings of files together with subspace's public key. Typically, SSKs are used to store indirect files containing pointers to CHKs rather than to store data files directly (like filenames and inodes in a conventional file-system).

In order to join the system a new node needs to know the address of any node already connected. The new node sends an announcement containing its identifying information (address, public key). The receiving node notes the received information and forwards the announcement to another node chosen randomly from its routing table. The announcement propagates until its TTL runs out. At that point, the nodes in the chain collectively assign the new node a random GUID in the keyspace using a protocol for shared random number generation that prevents any participant from biasing the result. This procedure assigns the new node responsibility for a region of keyspace that all participants agree on.

Freenet does not impose restrictions on the amount of data that publishers can insert. The system must therefore sometimes decide which files to keep and which to delete. Each Freenet node orders the files in its data store by time of last request, and when a new file arrives that cannot fit in the space available, the node deletes the least recently requested files until there is room. Because routing table entries are smaller, they can be kept longer than files. Deleted files do not necessarily disappear because the node can respond to a later request for the file using its routing table to contact the original data holder, which might be able to supply another copy.

Chapter 3

Model of Turtle network

In this chapter we introduce a system model of the Turtle network [1]. The Turtle network consists of a large set of nodes $\mathcal{N}$, and a large set of data items $\mathcal{D}$. We assume that behind each Turtle node i there is a human user (the node's owner) who has a subset D_i of all items in $\mathcal{D}$, and is interested in obtaining more. However, a user owning a node i is willing to share his data items only with nodes owned by people he trusts – we denote this as i's friends subset – F_i. We assume the friendship relation is commutative, for any two nodes i and j, if i is in F_j , then j is in F_i. However, friendship is not transitive (the friend of a friend is not automatically a friend).

Each data item d has an attribute set A_d associated with it. The attribute set consists of a number of $attribute = value$ pairs describing certain properties of the data item, and are used when evaluating user queries. These are logical expressions consisting of a number of $attribute = value$ pairs ($<, \leq, \geq, >$ allowed instead of $=$), connected using the AND, OR and NOT logical operators. A data item matches the query if the $attribute = value$ pairs in its attribute set satisfy the logical condition in the query expression.

Each user establishes a cryptographically secure connection between its Turtle node and all the nodes in its friends subset. Since there is no central trust infrastructure, the shared secrets needed to establish these secure connections have to be agreed-upon by out-of-band means (this can be done using common knowledge based on common past experiences – after all owners of friend Turtle nodes are assumed to be friends in the real life!). Once established, the inter-friends secure communication links are used for exchanging data items and propagating user queries.

[1] Parts of this chapter were taken from [26]

3.1 Querying in Turtle network

Users search for new data items by sending queries to the Turtle network. The user starts by introducing a query expression and a query depth through the query interface of its Turtle node. The node then generates a 64 bit random query ID. In this way, the probability that two distinct queries will have the same query ID is extremely small. Once it has the query ID, the node constructs a query packet containing the query expression, the query ID, and a time-to-live, initially set to the query depth. The query packet is then broadcast over the "friendship" links up to the desired query depth. Upon receiving a query packet, a Turtle node first evaluates the query expression against the attribute sets of all the data items in its data subset. If matches are found, the node reports them back to node that has forwarded (possibly originated) the query. Furthermore, the node decrements the time-to-live in the query packet, and if it is still positive, the packet is further forwarded to all the node's friends (except the one from which the packet came).

We can see that propagating a query in the Turtle network generates a *query broadcast tree* rooted in the node originating the query, tree that follows the trust relationships among the Turtle users. The query broadcast tree is also used for delivering query answers, which travel hop by hop up the tree until they reach the root. In order to match queries with answers, each node maintains a query table with queries it has forwarded recently. Each query table entry corresponds to a query broadcast tree the node is part of; table entries are indexed by query ID, and store the address of the node's parent in the tree and the time the query has been received. The table is also used for detecting *collisions*. A collision occurs when a node receives a query packet with a query ID that matches one already present in the query table. Since it is very unlikely that two different nodes will generate the same query ID, the most likely cause for the collision is a cycle in the friendship graph which has routed the same packet back. That is why such packets are ignored and not evaluated nor forwarded.

A query response packet consists of the address of the responder, the query ID, a response time-to-live, and a response payload consisting of data attribute set. A node receiving a query matching an element in its data set creates an answer with the payload consisting of the attribute set of the data item that matches the query. A node receiving a query hit from one of its children in the query broadcast tree will immediately report it to the parent. Before the query hit packet is forwarded, its time-to-live is decreased and the responder's address is changed to address of the forwarding node.

The query completes after the originating node collects reasonable number of answers from its friends or when user of the node does not want to

wait longer. The node then sorts through all these partial answers to identify all distinct data attribute sets; these are then presented to the user, much in the same way as the results of a web search engine query (they can be ranked based on frequency, or hop distance). Once the user selects the result he is interested in, the node can start the data retrieval phase.

The retrieval phase consists of selecting a retrieval path and propagating the query result along that path. For a given data element d (identified by its attribute set A_d), the retrieval path is the shortest path in the query tree between the root and a node that has d. This path is determined hop by hop, starting with the root node which searches through all response packets and selects the one that has A_d in the payload. The root then asks the friend from which the selected response packet has been received to retrieve the A_d data item. The friend follows a similar procedure to find the next hop in the retrieval path, and so on until the retrieval request reaches the node that has the actual data item. The data item is then sent to the requester, following (hop by hop) the retrieval path in reverse order.

3.2 Assumptions

There are two basic assumptions we make when proposing the Turtle architecture. First, we assume that continuous, high-speed Internet connections will become ubiquitous in the near future. Looking at current trends, which show increasing DSL/cable modem penetration in the consumer market, not to mention the everincreasing wireless "umbrellas" that cover large parts of big cities, this first assumption seems very reasonable.

The second assumption is that for sufficiently large social communities (a college campus, a country, the world), friendship relationships form fully connected graphs. Validating this assumption would obviously involve large-scale sociological experiments, which are beyond the scope of this thesis. However, based on the the moderate success of the PGP infrastructure, and, more recently, the explosive popularity of the Friendster[17] and Orkut[25] services, we have reasons to believe that for relationships involving moderate amount of trust, it is very likely to achieve full friendship graph connectivity. Detailed analysis of friendship graph of Orkut users is presented in Chapter 6.

3.3 Security of Turtle

From a security point of view, the Turtle architecture raises a number of interesting points:

First of all, Turtle offers good query initiator and query result anonymity: both initiator and responder are known only by their respective friends subsets (trusted nodes). With small modifications in the query/result routing protocol – namely removing the time-to-live field – it is also possible to achieve complete sender/receiver anonymity. Because all information exchange is done over encrypted channels, the only way for an adversary to link a query initiator to a responder is through traffic analysis. However, there are well known techniques for protecting against traffic analysis[15, 27], which can be easily incorporated in our basic query/result routing protocol.

Second, the Turtle network is immune to the "Sybil" attack[10]. Even if a powerful adversary is able to create a large number of malicious Turtle nodes, the effect these nodes have on the correct functioning of the system is minimal, unless the attacker is also able to infiltrate his nodes in the friends sets of correct nodes (but this would require a lot of social engineering!).

Third, Turtle exhibits a very desirable fail-mode property – "confined damage" – meaning that a security break in one correct Turtle node only affects a small subset of all correct nodes in the system (in this case the node itself plus its friends subset).

Finally, due to the way the Turtle overlay is organized, denial of service attacks typical for a peer-to-peer network – such as malicious routing [4], content masquerading (content that does not match its description), bogus query hits (a node answering positive to a query even when it does not have any matching content), and aborted transfers – are much less likely to happen. Because all direct interactions take place between nodes controlled by people who trust and respect each other (friends), we expect incentives for random malicious behavior to be very much reduced.

Chapter 4

Architecture and protocol specification

In this chapter, software architecture of Turtle node is presented. The description covers mostly communication infrastructure, because other services are either provided by the giFT framework (e.g. GUI) or they are not interesting (e.g. configuration management). The communication protocol used in Turtle network is described here as well.

4.1 Overview

Turtle nodes form a peer-to-peer network described in the previous chapter. Nodes are interconnected via TCP/IP – there is one TCP connection between two neighbours (unlike Gnutella[14], where separate TCP connections are opened for downloads, or FastTrack[12], where UDP packets are used for delivering small messages). Communication between any two entities in the Turtle network is done via virtual circuits that are tunnelled through TCP connections. Thousands of virtual circuits may be tunnelled through one TCP connection. On the other hand, one virtual circuit may consist of many *segments* (Figure 4.1). Higher level protocols use virtual circuits as their delivery mechanism.

There are five main components in Turtle (Figure 4.2):

- *TCP Channel Manager* is responsible for communication with neighbour nodes. It establishes secure (authenticated and encrypted) TCP connections to neighbour nodes and transfers data over these connections.

- *Router* is a central component of Turtle. All data going through the

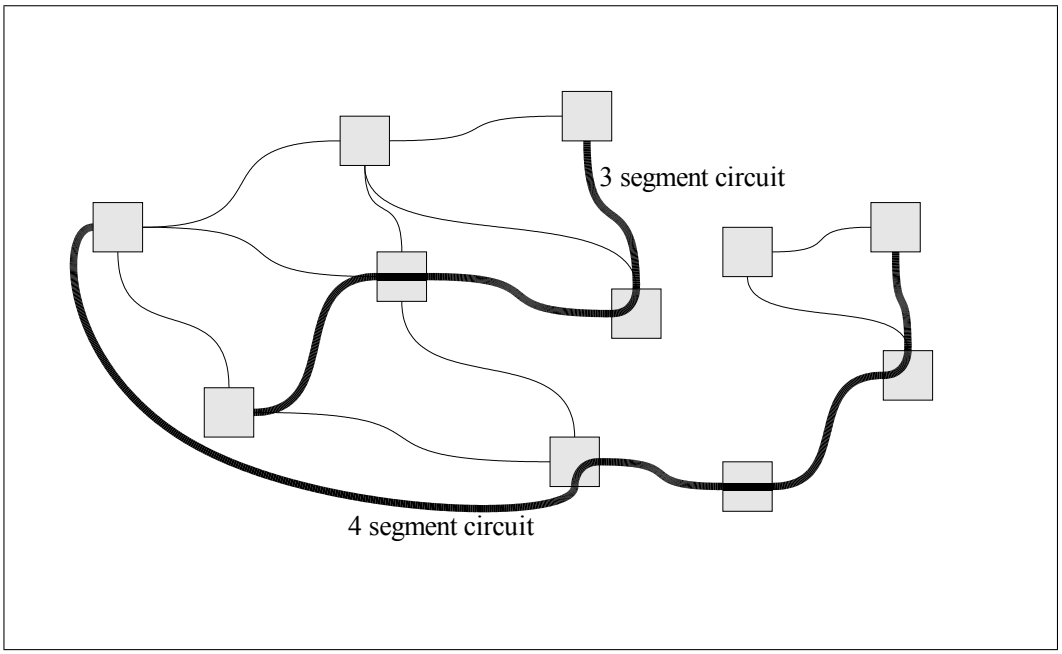


Figure 4.1: An example of virtual circuits in a small network of Turtle nodes.

node go through Router. Other components of Turtle are registered at Router and interact with it via *Channel* interface. We will call such components *channels*. Channel interface provides mostly methods for managing virtual circuits (opening, closing, sending data, controlling flow). Circuit switching between channels is done in Router.

- *Service Address Resolver* (or *SAR*) provides general mechanism for discovering services running at neighbour nodes.
- *Query Service* is responsible for receiving queries from neighbour nodes and from local GUI, evaluating them and forwarding them to neighbour nodes. It is also responsible for receiving query hits from neighbour nodes and forwarding them to the source of corresponding query.
- *File Service* acts as a simple file server, that allows others to download files from our node, and as a client for downloading files from other nodes.

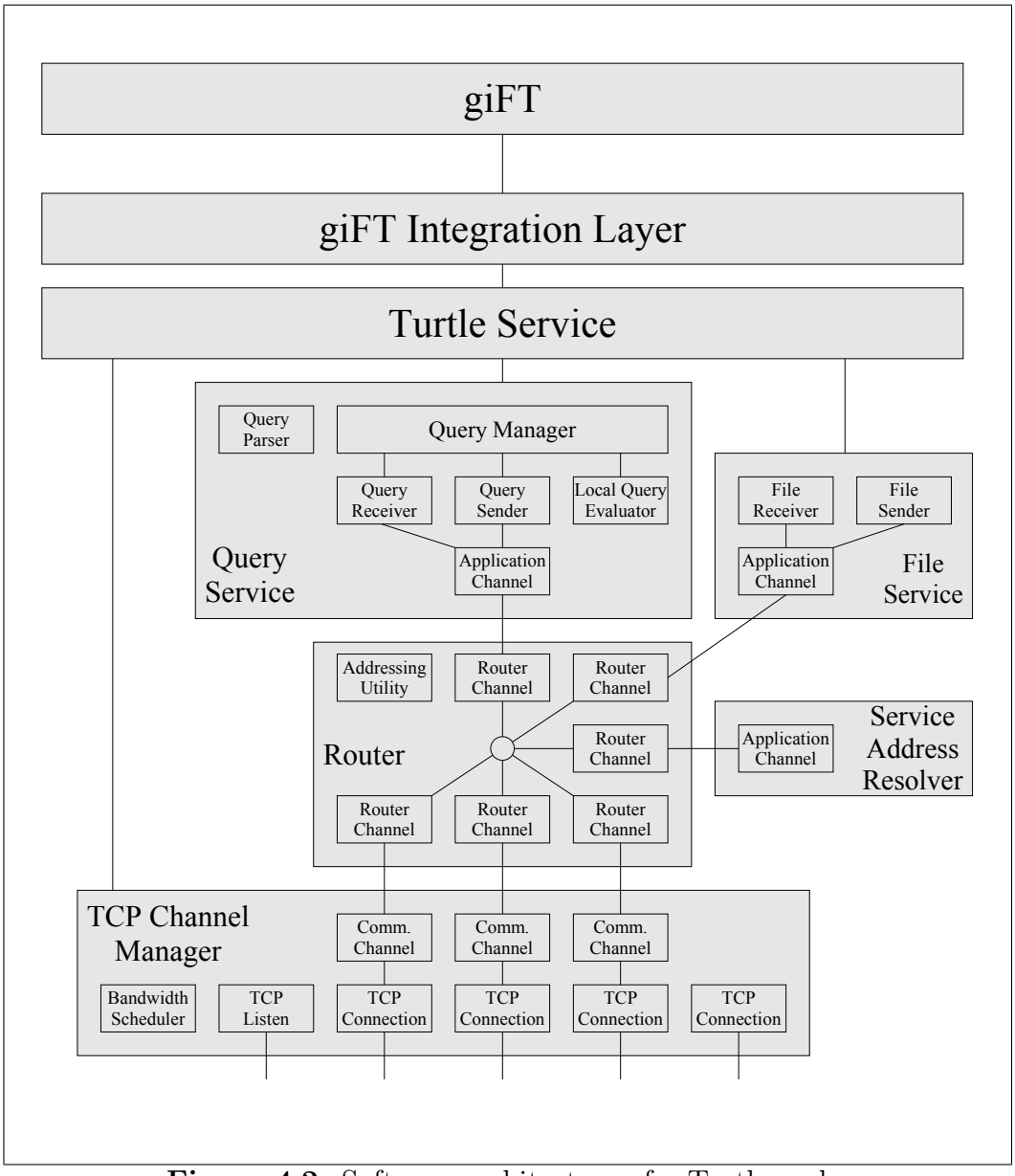


Figure 4.2: Software architecture of a Turtle node.

4.2 Communication with neighbour nodes

Turtle nodes are interconnected via TCP/IP connections. Each node maintains a set of neighbours – trusted nodes owned by friends. The set is configured by the user and is not dynamically updated from the network as in Gnutella[14] or in Freenet[16]. To add a new neighbour the user must enter neighbour's ID, IP address and secret master key. The same must be done by the other side, otherwise the nodes cannot establish connection.

TCP Channel Manager with its subcomponents is responsible for all communication related tasks. It accepts TCP connections coming from neighbours and also periodically tries to contact them. If a TCP connection is established, authentication procedure begins. During authentication procedure, two session keys (one for each direction) are generated and securely exchanged using shared master key. Data streams are then encrypted with session keys. Immediately after authentication a *Communication Channel* object is created and registered at Router. From that moment it is possible to create virtual circuits through newly created channel.

When TCP Channel Manager receives data from a TCP connection, it passes the data to corresponding Communication Channel. Communication Channel unmarshalls commands from the data and passes them to Router (using Channel interface). The inverse situation looks as follows. When Communication Channel receives command from Router, it marshalls the command to a buffer and passes it to TCP Channel Manager. TCP Channel Manager adds MAC (Message Authentication Code), encrypts everything and sends it to the neighbour node.

Because multiple virtual circuits may be tunnelled through one TCP connection, data multiplexing is necessary to guarantee fair division of connection bandwidth. This task is up to Communication Channel. It cyclically sends data of circuits and assigns higher priority to commands (CONNECT, FLOW CONTROL, CLOSE, etc.). Because network bandwidth is limited, Communication Channel might not be able to send data fast enough. That is why it has to control outgoing data flow of virtual circuits. Each circuit is supplied with send buffer of certain size. Flow control messages are sent to Router to ensure that this buffer is not overfilled. Section 4.3 gives more details about circuit flow control. Figure 4.3 shows a simplified picture of how Communication Channel works.

4.2.1 TCP connection initialization

TCP connection initialization is a four step procedure. The first three steps mutually authenticate newly connected nodes using challenge-response pro-

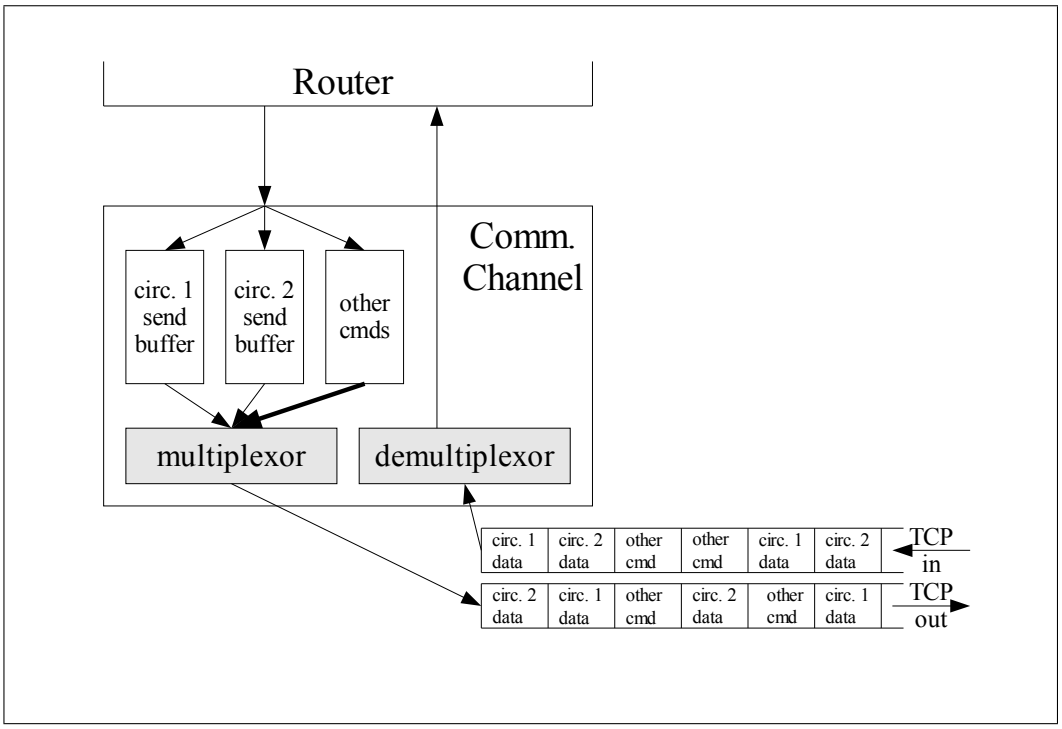


Figure 4.3: The Communication Channel component.

tocol. At the end of authentication procedure, nodes share the following information:

- Node ID of neighbour.
- Session key for outgoing data encryption.
- Initialization vector for outgoing data encryption.
- HMAC key for outgoing data authentication.
- Session key for incoming data decryption.
- Initialization vector for incoming data decryption.
- HMAC key for incoming data authentication.

The last step *activates* the connection. The activation is necessary, because neighbour nodes are equal and both may decide to connect to the other node at the same time. If this situation occurs, there are two equal TCP connections, but only one of them may survive. During the activation step the node with *the higher node ID* decides whether the newly authenticated connection should survive. Because the responsibility for activating

the connection lies always on the same node no matter who initiated the connection, it never happens that both connections are activated or that both connections are closed.

The first authentication packet is sent by the initiator of TCP connection (node **A**) to the node that accepted the connection (node **B**). Table 4.1 shows its structure.

Length	Description
47	ID of node **A**
16	Random number generated by **A**

Table 4.1: The first authentication packet (A → B).

After receiving the first packet node **B** knows identity of node **A**. Node **B** then searches its keystore for the master key shared with this node. If the master key is not found, the connection is immediately closed. Otherwise node **B** sends the second authentication packet (Table 4.2). The whole packet is encrypted with AES[1] in CBC mode using the master key.

Length	Description
16	Random number from the first packet
47	ID of node **B**
16	Random number generated by **B**
16	AES session key for data stream **A** → **B**
16	AES initialization vector for data stream **A** → **B**
16	HMAC key for data stream **A** → **B**
20	HMAC/SHA1 digest of bytes 0–126
13	Zero padding

Table 4.2: The second authentication packet (B → A).

Node **A** can now verify node **B**'s identity. It decrypts the second authentication packet with master key and checks the following conditions:

- Random number at the beginning of the packet must be same as the number in the first authentication packet (to guarantee freshness).
- ID of node **B** must be same as what is configured at node **A**.
- HMAC digest of the packet must be correct. The digest is computed with its key set to the master key.

During the last step node **A** authenticates itself to the node **B**. The third authentication packet (Table 4.3) is again encrypted with AES in CBC mode using the master key.

Length	Description
16	Random number from the second packet
16	AES session key for data stream **B** → **A**
16	AES initialization vector for data stream **B** → **A**
16	HMAC key for data stream **B** → **A**
20	HMAC/SHA1 digest of bytes 0–63
12	Zero padding

Table 4.3: The third authentication packet (A → B).

Node **B** decrypts the third authentication packet and performs same checks as node **A** in the previous step. After successful authentication both nodes have all information needed to initialize encryption, decryption and MAC for outgoing and incoming data streams. Before starting the real data transfer the connection must be activated. It is done by the node with the higher node ID, which sends one byte containing zero to the node with the lower node ID.

4.2.2 Data transfer

Data stream sent via TCP connection is divided into *blocks* (Table 4.4). Each block begins with header, which contains length of the data being transferred in this block. The header is followed by data themselves, HMAC/SHA1 digest of the data and padding. Everything but the header is encrypted with AES in CBC mode. Both AES and HMAC are initialized only once immediately after authentication, and not before each block.

Length	Description
4	Length of data
1–8192	Data
20	HMAC digest of data
0–15	Zero padding

Table 4.4: Structure of data block transferred via TCP connection.

4.3 Virtual circuits

Virtual circuits are used to transfer data between any two entities in the Turtle network (e.g. Query Service, File Service or Service Address Resolver). Virtual circuits are tunnelled through secure TCP connections described in Section 4.2. Virtual circuit may consist of one or more segments. If it only goes from a node to its neighbour, then it consists of one segment. If it goes to a neighbour and then to neighbour of this neighbour, it has two segments. Maximum number of segments of one circuit is theoretically not limited, but in practise it will be less than the diameter of the network graph, which will be low.

4.3.1 Circuit life-cycle and data flow

In this section, we describe commands (messages) that control life-cycle and data flow of virtual circuits. The description is simplified to a circuit with one segment. Details about circuits with more segments will be given in Section 4.3.2.

There are six basic commands related to virtual circuits:

- CONNECT is a request to create a new circuit. It is always the first command of the circuit.
- CONNECTED is a reply to CONNECT command. It is sent when the circuit has been successfully created.
- CLOSE is a request to close the circuit. It may be sent as a reply to CONNECT command, if the circuit cannot be created.
- CLOSED is a reply to CLOSE command. It confirms that the circuit has been closed. It is always the last command sent to the circuit.
- FORWARD command transfers data through the circuit. It can only be sent on established circuit – after CONNECTED command is received or sent and before CLOSE command is received or sent.
- FLOW CONTROL command prevents congestion of data receiver, if it cannot process received data fast enough. It can only be sent on established circuit. Flow control command gives certain amount of *credits to send more data*. Each credit permits receiver of this command (i.e. data sender) to send one more byte.

Figure 4.4 shows an example of life-cycle of a virtual circuit. In this example, node **A** acts as a client that connects to node **B**, sends short data (e.g. a HTTP request), receives long data (e.g. a HTTP reply) and closes the circuit.

The interaction begins when node **A** sends CONNECT command. Node **B** accepts the circuit and replies with CONNECTED command. Circuit is now established for node **B**. CONNECTED command is immediately followed by FLOW CONTROL command with 500 credits, because node **B** has 500 bytes long receive buffer. The circuit becomes established for node **A** as soon as it receives CONNECTED command. Node **A** sends FLOW CONTROL command with 500 credits, because it has 500 bytes long receive buffer, and then waits for credits from node **B**. After receiving FLOW CONTROL command node **A** sends FORWARD command with 100 bytes of the request. The request cannot be sent sooner, because node **A** does not know, whether node **B** is ready to receive data.

The request arrives to node **B** and is processed. Because the reply has 700 bytes, it cannot be sent all at once. That is why node **B** sends only 500 bytes, waits for more credits and sends the remaining 200 bytes. Node **A** closes the circuit after it receives the whole reply. First, it sends CLOSE command. Node **B** replies with CLOSED command and forgets the circuit. Node **A** waits for reception of CLOSED command and then forgets the circuit.

The reason for having separate CLOSE and CLOSED command is that it gives us exact moment when the circuit can be forgotten, because no more circuit commands may arrive. Without CLOSED command, it would be possible to ignore commands for unknown circuits, but it has two disadvantages. First, circuit IDs could not be reused. And second, it is less immune to errors in implementation. Figures 4.5 and 4.6 show situations when a command arrives after CLOSE command has been sent.

There are two more commands not mentioned before:

- CIRCUIT CONTROL command gives *credits to open more circuits*. Each newly created circuit costs one credit. Nodes are not allowed to open more circuits than how many circuit credits they have at the moment. This command protects nodes from being overloaded by too many incoming virtual circuits.

- SAR ADDRESS command informs about new address of the Service Address Resolver. This command should be sent very early after the TCP connection has been established. See Section 4.4 for more details.

Exact structure of all command packets is given in Appendix A.1.

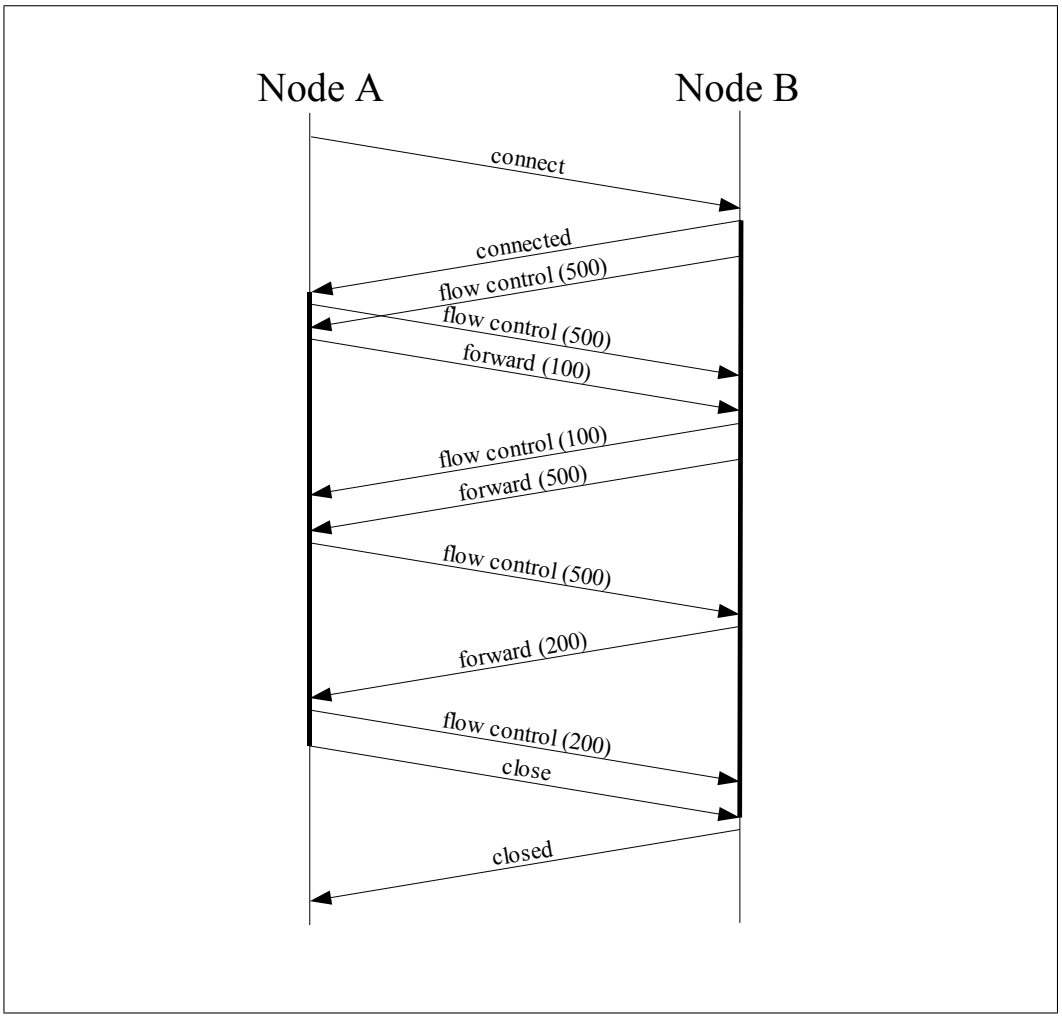


Figure 4.4: An example of life-cycle of a virtual circuit.

4.3.2 Command routing

In this section we will go more into details of how virtual circuits are created and how data flow through the Turtle network. We have introduced CONNECT command that creates a circuit, however there was no mechanism to specify end point of the circuit. The solution is not surprising – *addressing*.

Each CONNECT command comes with the source address and the target address. Source address is not very important at this moment, so let us concentrate on the target address. Target address specifies the end point of the circuit – either a service on the local node or the end point can reside at a different node. Whenever a node receives CONNECT command from its neighbour, it examines the address, retrieves information about next hop and forwards the CONNECT command to appropriate neighbour or local service.

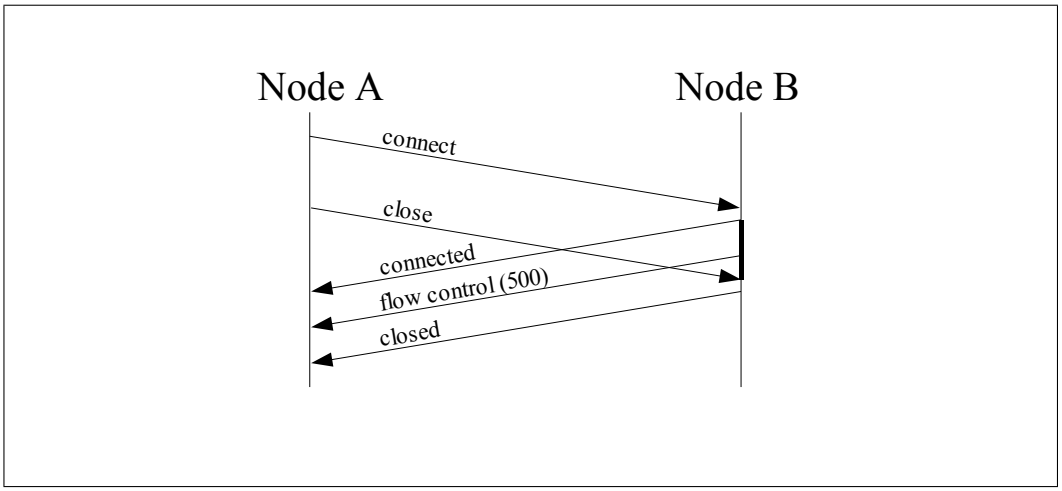


Figure 4.5: An example of a virtual circuit that is prematurely closed by its initiator.

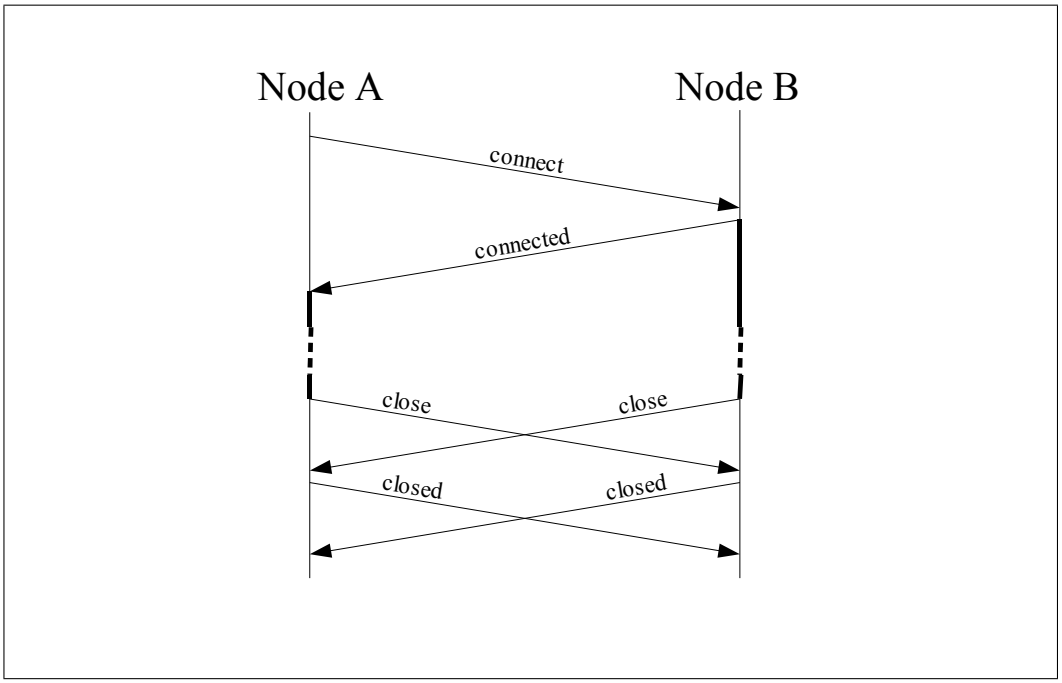


Figure 4.6: An example of a virtual circuit that is closed by both nodes at the same time.

A circuit with multiple segments is established hop-by-hop in this way.

Figure 4.7 shows a virtual circuit with 3 segments. CONNECT command is forwarded from node **A** via nodes **B** and **C** to node **D**. CONNECTED command goes the opposite direction. FLOW CONTROL commands immediately follow CONNECTED command, because all nodes on the path provide the circuit with buffer, so they control data flow on their adjacent circuit segments. This is different from TCP/IP, where data flow is controlled by connection end points. The FORWARD command in the figure transfers data along established path. It generates a FLOW CONTROL command at every node. FLOW CONTROL command is sent when the data leave the node making more space in circuit buffer. In our example node **D** does not reply and immediately after reception of data closes the circuit. CLOSE command is again propagated hop-by-hop, followed by CLOSED command.

Different example of virtual circuit is shown in figure 4.8. The circuit is established in the same way as in the previous example, but this time node **D** crashes after a while. Node **C** detects the crash (TCP connection goes down) and initiates circuit shutdown procedure. Notice that node **B** does not forward data received from node **A**, because it has already received CLOSE command from node **C**.

Let us now look what happens when circuit commands arrive at a node. First, they are processed by TCP Channel Manager as described in Section 4.2. Then they are passed to the Router component via Channel interface. Router maintains a table of opened circuits for each channel. Each circuit has a record in the table saying where it continues – to which channel and what is the circuit segment ID at the target channel. Based on this information router decides, where to forward the command. It passes the command to selected channel, which undertakes responsibility for the command.

There are two types of channels that can be registered at Router. One of them is above mentioned Communication Channel. The second one is *Application Channel*, which has the role of end point of virtual circuits. For Router there is no difference between Communication Channel and Application Channel. Both have the Channel interface, so they accept and invoke CONNECT, CONNECTED, etc. commands. Router does not see that Communication Channel communicates all commands with a neighbour node, while Application Channel processes everything locally. The reason to put Application Channel component between Router and a service is that it has easy to use, BSD sockets like, interface. It makes development of Turtle services easy for programmers with knowledge of BSD sockets library.

Application Channel provides every circuit with receive and send buffer. Together with Communication Channel it controls data flow of virtual circuits. Figure 4.9 shows an example of a two segment virtual circuit with

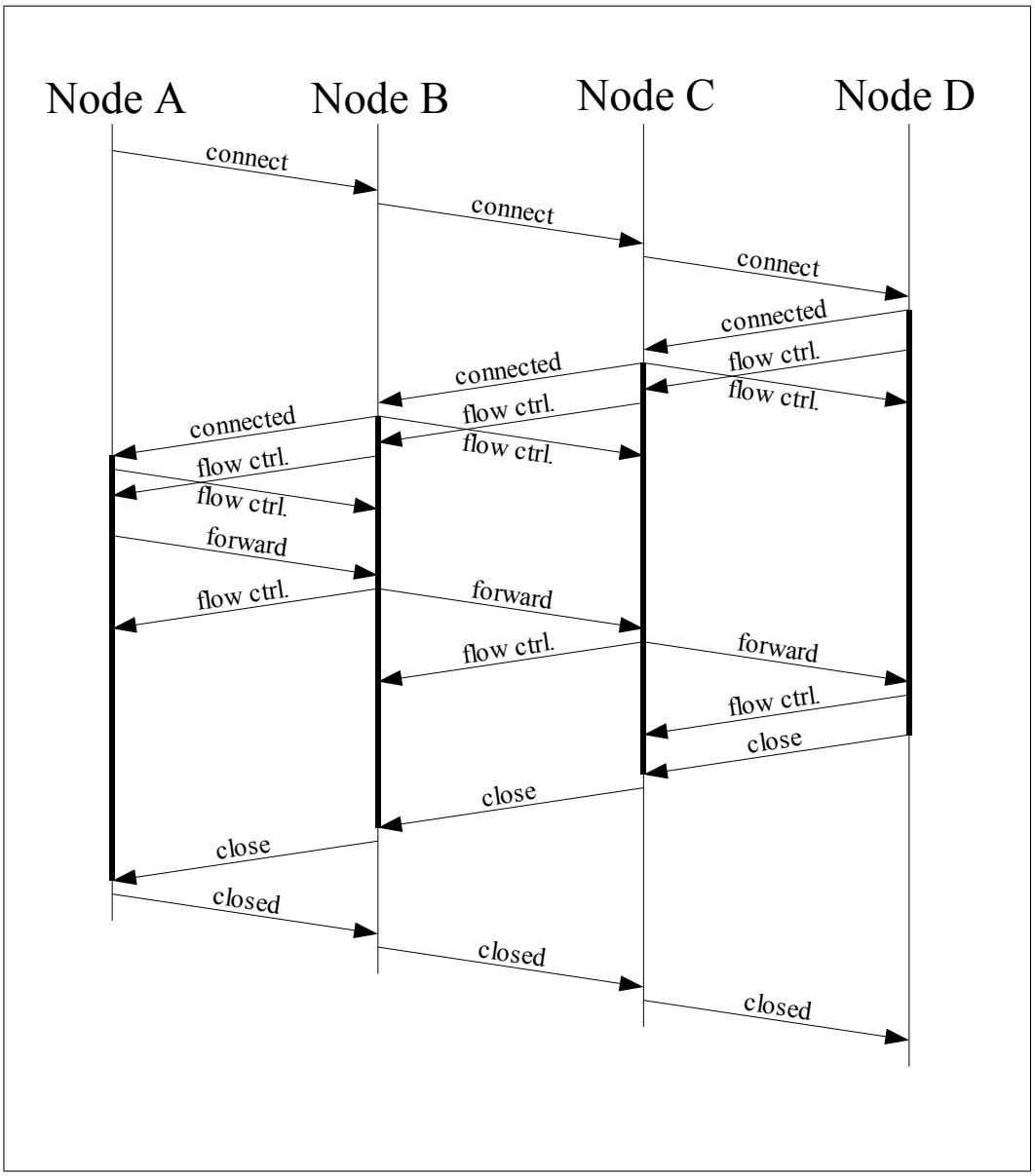


Figure 4.7: An example of a virtual circuit with 3 segments.

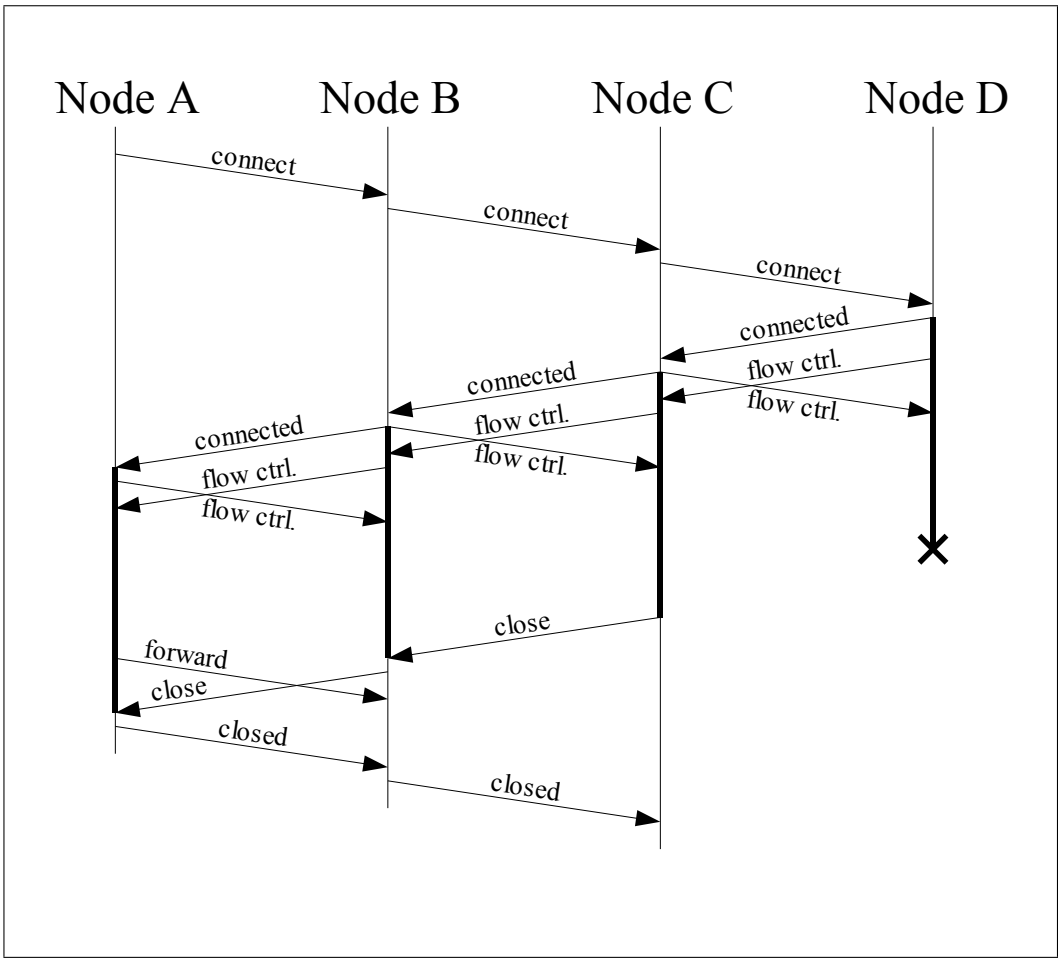


Figure 4.8: An example of a virtual circuit with 3 segments, where one node crashes.

circuit buffers symbolized by small black squares. They are always at places where data might get stuck for some reason – either because of slow link (in Communication Channel) or because of slow application (in Application Channel). Every circuit buffer is not only source of FLOW CONTROL commands, but it has also one side effect. When a FORWARD command reaches a send buffer, data from the command are put into the buffer. Later, when they are sent further, new FORWARD command is created for them. Size of chunks, in which data are sent, is independent on how those data were received. That is why send buffers cause splitting and merging of FORWARD commands.

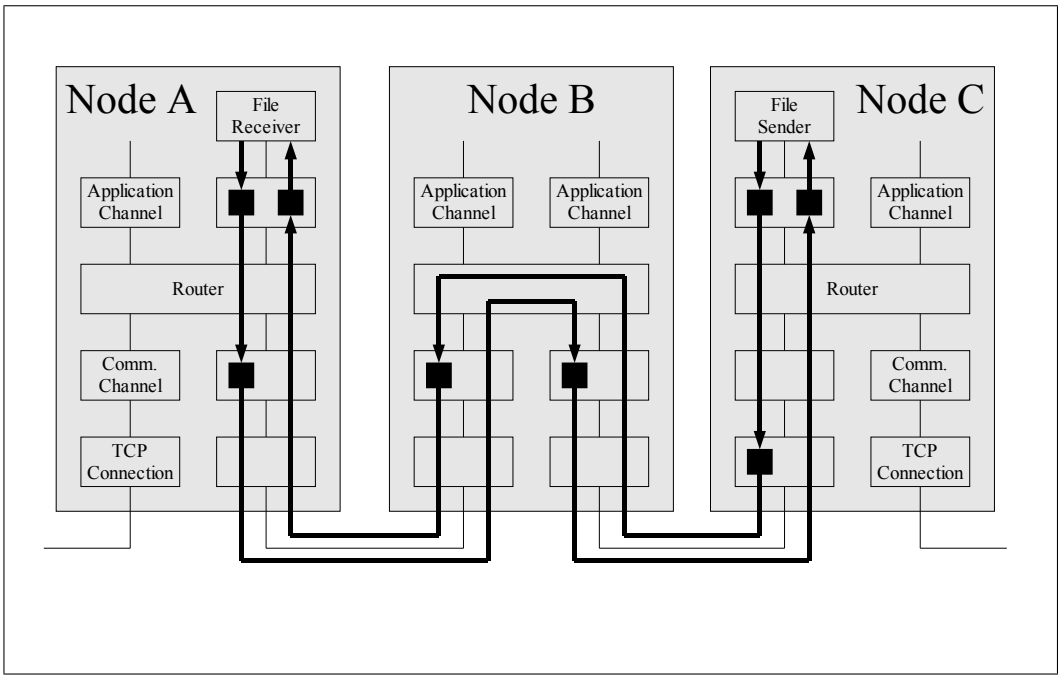


Figure 4.9: An example of a virtual circuit with demonstration of flow of data through Turtle components. Black boxes represent buffers.

4.3.3 Addressing

In the previous section we described, how circuit commands flow through Turtle node. Although not mentioned, it is obvious, that CONNECT command needs special handling at Router. When Router receives the CONNECT command, circuit is not established yet and Router does not know, where to forward the command. It has to parse the target address to extract *routing information*, which currently consists of ID of channel, where the circuit continues. In the future, routing information might also contain something else, e.g. QoS information. When Router retrieves the routing information, it updates its data structures and forwards the CONNECT command to proper channel. Forwarded command contains *updated target address*, so that receiver can again parse it in order to get its own routing information. In the following sections we will describe two different addressing schemes, each suitable for different purpose. The Router should be able to work with both of them, although the second one is not mandatory. Both addressing schemes try to hide identity of end point of the circuit, so we refer to addresses created by them as to *anonymous addresses*.

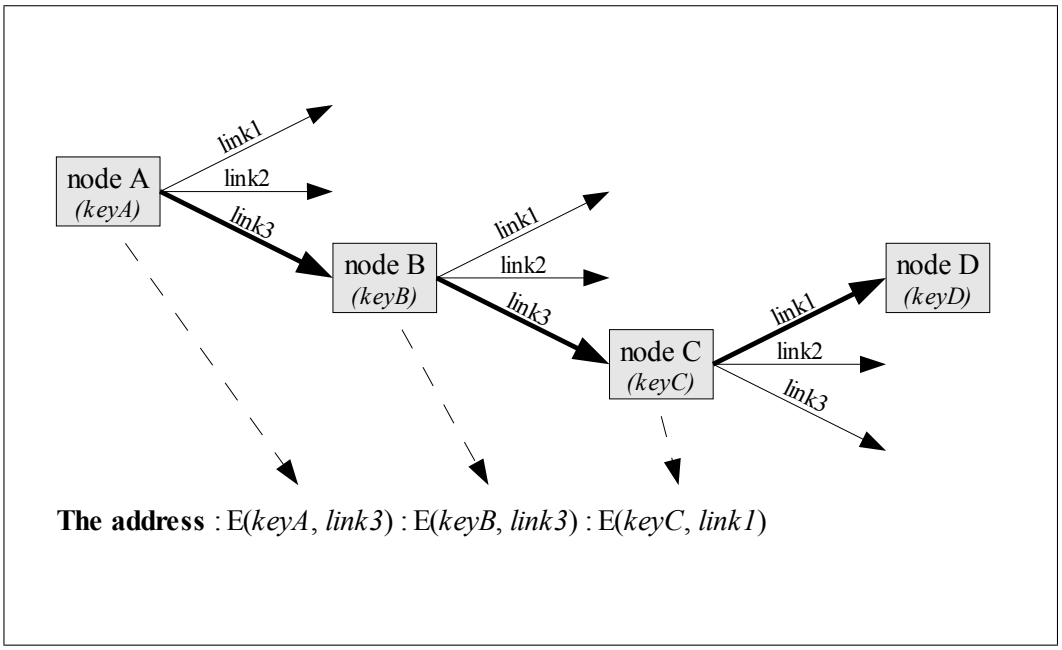


Figure 4.10: Construction of stateless address.

Stateless addressing

Stateless addressing is called stateless, because it does not require nodes along the path from source to target to store any state. The whole routing information is stored in the address – each hop extends the address by a short *record*, saying, where the circuit continues. This technique is known from source routing, however in Turtle it is used with one major modification. Each record is encrypted by the node that later extracts it from the address, when circuit is being established. This has the advantage that nobody else then the creator of the record understands it.

Figure 4.10 gives an example of a very simple network consisting of four nodes. Each node keeps an encryption key used for encrypting anonymous addresses. Then the address of node **D** at node **A** is E(*keyA*, *link3*):E(*keyB*, *link3*):E(*keyC*, *link1*). Because nodes understand only their own part of the address, they know only their immediate neighbours on the path. This addressing scheme also reveals path length to nodes. Although it does not look like very useful information, it might become dangerous when path length is low (1 or 2 hops). That is why stateless addressing is now used only when hop count does not need to be kept secret.

Anonymous addresses are usually constructed on the way from the target node back to the source node (Section 4.5 gives more details). For stateless anonymous address it means that each node extends the address by *prepend-*

	Length	Description
	4	Address type (2)
for each record (encrypted)	4	Random data
	4	Type of record
		(0 normal, 1 if enc. address follows)
	1	Length of record data
	0–255	Record data
	0–15	Zero padding
encapsulated address (encrypted, optional)	4	Random data
	4	Type of record (2)
	4	Length of encapsulated address
	4–	Encapsulated address
	0–15	Zero padding

Table 4.5: Recommended structure of stateless anonymous address.

ing node's own routing information in encrypted form. Because the routing information is later interpreted by the same node, there are no limitations on the structure of prepended data. That is why the only requirement is that the whole address begins with proper address type value, everything else are just recommendations.

Table 4.5 shows recommended structure of the address. As an encryption method, AES is recommended. Encryption key does not need to be persistent, so the address becomes invalid after any node on the path is restarted. The most important part of the address is record data field, where routing information is stored. Currently it contains only channel ID. The address may optionally end with special record called *encapsulated address*, which allows to combine different addressing methods in one address, e.g. first two hops of five hop address use stateless addressing and remaining three use stateful addressing.

Stateful addressing

Stateful addressing requires every node along the path from source to target to store a piece of routing information. Routing information is stored in a big table and stateful anonymous address contains only index (somehow obfuscated) to this table. This approach has an advantage that all addresses have the same length no matter how far it is to the target node. The disadvantage is that routing information must stay in the network for some time, which consumes memory of nodes. Because there is no mechanism to tell nodes

Length	Description
4	Address type (3)
8	Record ID

Table 4.6: Structure of stateful anonymous address.

that certain address is not useful any more, nodes must forget old addresses to make place for new ones. There is no strict rule about how long the Turtle nodes must keep addresses in their memory. We recommend to keep them at least 5–15 minutes, because stateful addresses are currently returned by Query Service, so they should not be forgotten before user decides to download a file.

Here is the recapitulation of advantages and disadvantages of stateful anonymous address against stateless anonymous address:

+ It does not reveal hop count.

+ It is shorter (less communication overhead).

− It consumes memory of all nodes on the path.

− It spontaneously becomes invalid after some time.

Structure of stateful anonymous address is given in Table 4.6. Record ID might be anything that identifies a record in the table of addresses. Each record of the table contains channel ID and record ID for the next node on the path. It should not be easy to guess a valid record ID, because malicious nodes could try to connect to addresses that were not created for them. Generating record IDs randomly is sufficient solution. The problem does not even appear if there is separate table for each neighbour node.

4.4 Service Address Resolver

Service Address Resolver (SAR for short) provides mechanism for discovering services running at neighbour nodes. It is used by Query Service, which needs to communicate with neighbour nodes in order to exchange queries and query hits. However, SAR is general mechanism and might be used by any other service as well.

There is one SAR running on each node. It is registered at the Router as a regular service with special flag set. The flag tells the Router to *send address of this service to all neighbours*. The address is sent via SAR ADDRESS

command immediately after the TCP connection to a neighbour has been established. Because it is the only address sent initially to neighbours, it acts as an entry point to the node. Any service at neighbour node may open a virtual circuit to our SAR and request the following information:

- *List of services.* Gives list of names of services that registered themselves at SAR, e.g. Query Service (File Service is not registered at SAR – its address is provided by Query Service instead).
- *Address of a service.* Gives anonymous address of requested service. This request is used by Query Service – it contacts neighbour's SAR and asks for anonymous address of Query Service running at that node.

Exact structure of SAR packets is given in Appendix A.2.

4.5 Query Service

Query Service implements querying scheme described in Section 3.1. When a query is entered by a user, it is first parsed to check correct syntax. A *parse tree* is constructed and used to build a query packet. Although using parse tree instead of original query string might slightly increase size of query packet, it reduces CPU load of nodes in the Turtle network, because parsing is done only once and not on every node of query broadcast tree. Query packet is then inserted into the Turtle network and is propagated until the desired maximum depth.

Queries are sent using the *Query Sender* component. Query Sender maintains set of virtual circuits permanently connected to all neighbour nodes. When a new neighbour appears, Query Sender contacts neighbour's SAR and asks for address of *Query Receiver* running at that node. Then a virtual circuit is opened to Query Receiver. The circuit is used for sending queries and receiving query hits. For the other direction of communication – receiving queries and sending query hits – there is second circuit initiated by neighbour's Query Sender, which is connected to our Query Receiver. Having two circuits with simple protocol instead of one with complex protocol makes implementation easier.

When Query Receiver receives a query from a neighbour node, it passes the query to *Query Manager*. Query Manager is responsible for detecting collisions caused by cycles in friendship graph. For this purpose, it maintains table of recently seen queries. If the received query is found in the table, it means collision, and the query is ignored. Otherwise the query is passed to Query Sender, which forwards it to all neighbours except the one that

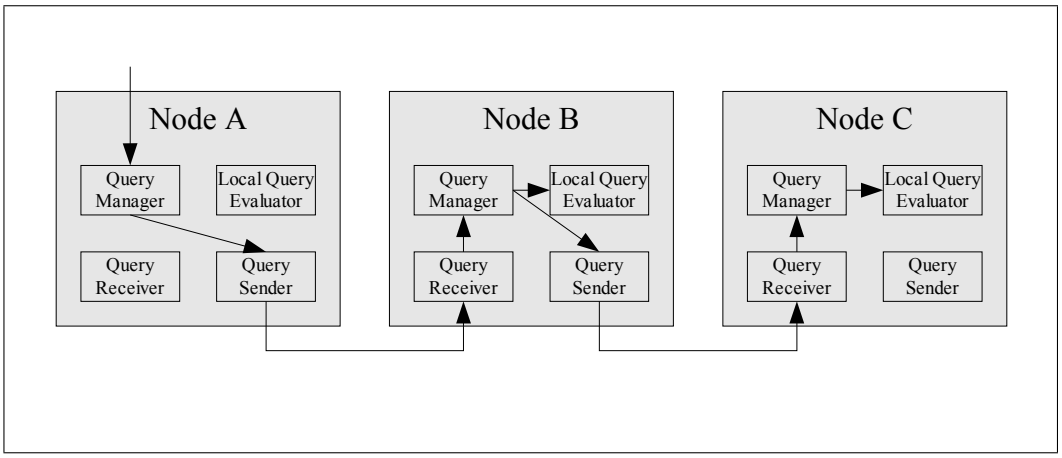


Figure 4.11: An example of how a query is propagated through a small Turtle network

forwarded it to us. The query is also passed to *Local Query Evaluator*, which evaluates the query against locally shared files.

If Local Query Evaluator finds a hit, it builds a query hit packet that contains all attributes of the hit (file name, file length, etc.), anonymous address of local File Sender and query ID of corresponding query. Query hit packet is passed to Query Manager, which finds query ID in the table of recently seen queries. The table also contains query source for each query. With this information, Query Manager passes the query hit packet to Query Receiver, which sends it to appropriate neighbour via one of the permanently connected virtual circuits

When Query Sender receives a query hit packet from a neighbour node, it first *extends the File Sender anonymous address* in the packet by local routing information. How exactly is the extension performed depends on the type of anonymous address used. Currently stateful addresses (Section 4.3.3) are preferred for query hits, but stateless addresses could be used as well. Modified query hit packet is then passed to Query Manager, where it is handled in similar way as query hits received from Local Query Evaluator – it is forwarded either to Query Receiver or to local application.

Figures 4.11 and 4.12 give an example of how query packets and query hit packets are propagated through a small Turtle network. Exact structure of query packets is given in Appendix A.3.

4.5.1 Query syntax

The basic building block of a query is formula with attribute name, relational operator and a value (if the operator is not unary):

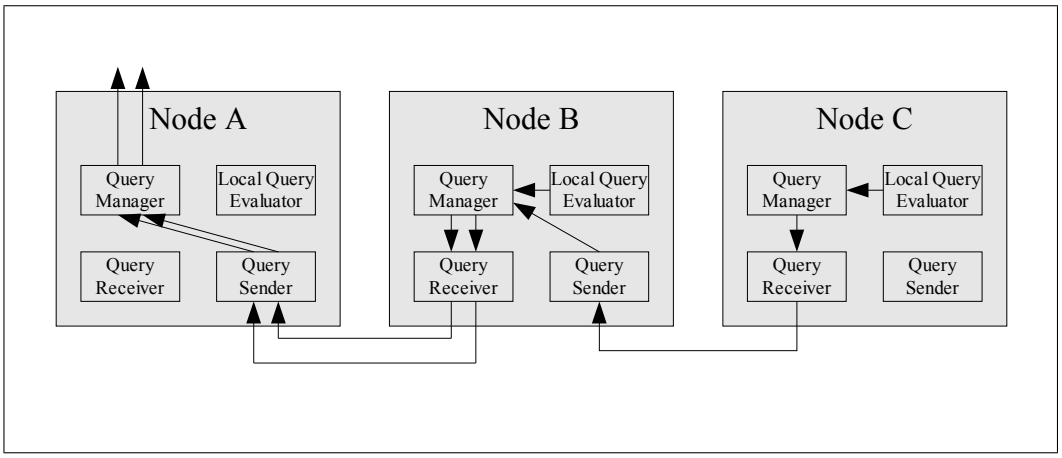


Figure 4.12: An example of how query hits are propagated through a small Turtle network

- *attribute* > *value* True if attribute is greater than value.
- *attribute* >= *value* True if attribute is greater than or equal to value.
- *attribute* == *value* True if attribute is equal to value.
- *attribute* <= *value* True if attribute is less than or equal to value.
- *attribute* < *value* True if attribute is less than value.
- *attribute* = *value* True if value is a substring of attribute.
- *`EXISTS` attribute* True if attribute is defined.

Neither attribute name nor value may contain spaces. However if value is enclosed in quotation marks or single quotes, spaces are allowed. Backslash is used as an escaping character to insert quotation marks or single quotes to value. String comparisons are case insensitive.

Basic formulae are connected with logical operators.

- *formula1* `AND` *formula2* True if both formulae are true. Ampersand (&) is a shortcut for `AND` operator.
- *formula1* `OR` *formula2* True if at least one of formulae is true. Pipe (|) is a shortcut for `OR` operator.
- *`NOT` formula* True if formula is false. Exclamation mark (!) is a shortcut for `NOT` operator.

More complex logical expressions can be built using parentheses. A few examples of queries follow.

- *name = turtle*
  True for file with name containing *turtle* (or *Turtle*, *TURTLE*, etc.).
- *name == turtledoc.pdf* `AND` *length > 100000*
  True for file *turtledoc.pdf* longer than 100kB.
- *name = turtle* `AND` *(type == pdf* `OR NOT EXISTS` *type)*
  True for file with name containing *turtle*, which has type *pdf* or has undefined type.
- *name = turtle* & ! *name == 'turtle w/ spaces and \'single quotes\'.avi'*
  True for file with name containing *turtle*, with one weird exception.

4.6 File Service

File Service is a service that transfers files over the network. It has two subcomponents – File Sender and File Receiver. When a user of the Turtle decides to download file from the network, he first uses Query Service to locate the file. Besides other attributes, Query Service returns name of file and anonymous address of File Sender of node where the file resides. The user then asks local File Receiver to download the file for him.

File Receiver manages all file downloads. When asked to download a file, it opens a virtual circuit to remote File Sender and sends request packet. The packet contains file name, starting position and maximum length of data. Remote File Sender finds the file on its filesystem and responds with the desired data. During the transfer all data received by File Receiver are stored to local filesystem to location specified by the user. After the transfer finishes, user is notified that his file is ready. Structure of packets used in communication between File Receiver and File Sender is described in Appendix A.4.

Protocol described here supports continuing of interrupted downloads, which is an important feature, because the chance that a given circuit in the Turtle network will be interrupted is much higher than in normal peer-to-peer networks. It happens whenever any node on the path is shut down. Circuits used to download long files are especially vulnerable to this.

When a download is interrupted, it is usually not possible to restart it by simply connecting to the same address as before, even when the target node is up. The reason is that the address routes the circuit through node that

is probably down. That is why a new query must be issued to find different route to the target node. After the node is found (or different node with the same file), the download can be restarted.

4.7 giFT Integration Layer

The giFT Integration Layer was added to Turtle after we found the giFT project[18]. The giFT framework, often referred to as the giFT bridge, is implemented as a separate daemon process to which protocol plug-ins can be added. The bridge provides a generic interface that can be used by client front ends to access any of the networks currently installed.

The giFT Integration Layer adapts Turtle to the giFT interface for protocol plug-ins, i.e. Turtle has become a protocol plug-in of giFT. The integration with giFT was quite straightforward (besides few unavoidable hacks) and as a results, Turtle gained many new features. The following list gives overview of the most important ones.

GUI Any giFT front end can now be used as a Turtle front end. There are several professionally looking graphical front ends, such as Apollon[3] or giFToxic[19]. The solution with generic front end suffers from one major drawback – the front end does not understand the underlying protocol specifics, so the configuration must be done by hand and not via user-friendly dialog boxes.

Metadata Most of the files that are published to the network can be provided with metadata based on the file content, e.g. author of a document or bitrate of a song. The ability to retrieve such information from a file requires knowledge of the file structure. The giFT framework scans all locally shared files, and reads metadata from files with known structure. It passes the metadata to protocol plugins. In our case, the metadata are passed to Local Query Evaluator, which incorporates them into query hit packets.

File fingerprints and hash-links Besides reading metadata from locally shared files, giFT also calculates their hashes (fingerprints) and passes them to protocol plugins. Turtle sends file hashes via query hit packets to query initiator, which can later verify that downloaded file is not corrupted. File hash can be also used to find a file by its content, if someone publishes a hash of the file (e.g. on the web).

Multisource downloading Downloading file simultaneously from multiple sources is another feature of giFT. It is completely transparent for

protocol plugins. giFT locates multiple sources of the same file via file hash (same file hash means same file even if its name is different) and initiates multiple downloads, each starting at different position. For Turtle, these downloads seem to be totally independent.

Different query syntax Queries produced by giFT have different syntax than Turtle queries. Because Turtle queries are more general, giFT queries can be easily translated into Turtle syntax without loss of information.

Chapter 5

Performance measurements

In order to verify that our implementation of Turtle is reasonably good and does not suffer from major performance problems, we ran set of performance tests. All tests were run on DAS2 machine[8], a cluster of 72 PCs located at Vrije University. DAS2 nodes have the following configuration:

Processors	2 x Pentium III, 1 GHZ
Memory	1 GB
HDD	20 GB, IDE
Network interfaces	Fast Ethernet (100 Mb/s) Myrinet-2000
Operating system	Red Hat Linux 7.3 with kernel 2.4.20-24.7smp

Table 5.1: DAS2 node configuration overview.

TCP stack of DAS2 nodes was configured to use Fast Ethernet network card, because drivers for Myrinet cards are optimized for userland libraries like MPI and they have high latency when used with TCP/IP. Fast Ethernet network of DAS2 is fully switched at 100 Mb/s with full-duplex (our tests showed that each node is able to send and receive almost 11 MB/s at the same time).

5.1 Protocol overhead

The aim of the protocol overhead test is to measure, what part of data transferred by Turtle is actual file data and how much control data is added by Turtle. The topology of Turtle network used in this test is depicted

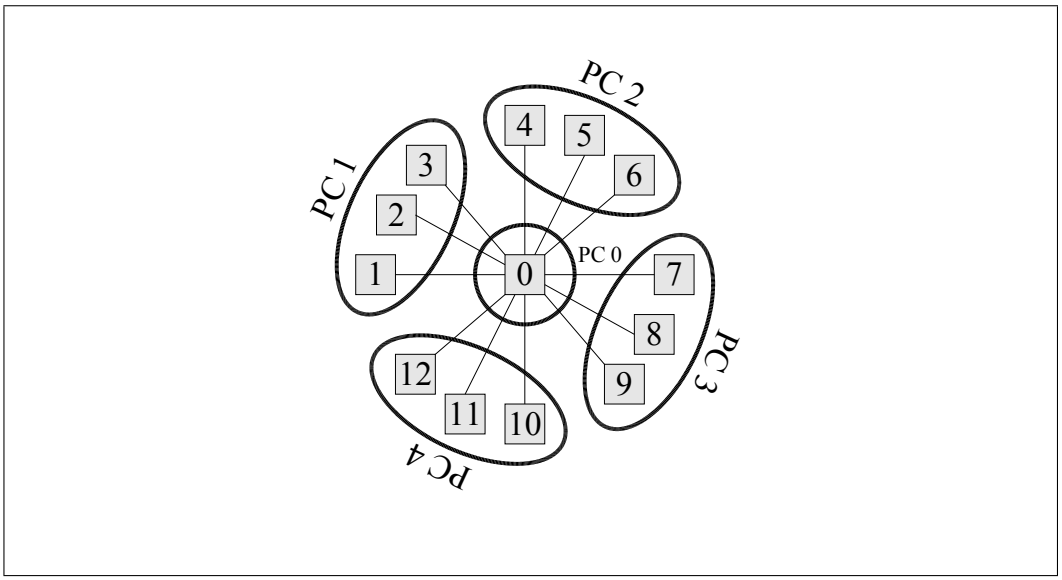


Figure 5.1: Protocol overhead experimental setup – star topology with node 0 in the middle.

in Figure 5.1. Node 0 in the middle relayed all traffic and performed the measurements. Other nodes just transferred huge files as fast as node 0 allowed them to do. Node 0 ran on a dedicated PC to minimize influence of CPU load, while leaf nodes shared PCs (there were more Turtle nodes than PCs, so we could not avoid it).

The measurement was done in the following way. First, all leaf nodes established TCP connection to node 0, which was their only neighbour. Leaf nodes then issued queries to find files located at other leaf nodes. After receiving all replies, each leaf node opened certain number of file download circuits to randomly chosen nodes and started downloading. Node 0 relayed the traffic and periodically (every second) recorded amount of file data relayed. Because bandwidth of node 0 was limited by configuration, we could easily compute protocol overhead as $(BW - F)/F$, where BW is bandwidth and F is file data volume.

Each test run took 10 minutes, however values from first 5 minutes and last 2 minutes were ignored. Values from 3 minutes in between were averaged and used in formula in previous paragraph. By using only values from the middle of the interval, the measurement is not influenced by "warmup period", when nodes establish TCP connections and virtual circuits, and communication buffers are being filled.

There are several data structures that are communicated in Turtle protocol and that cause the overhead:

Bandwidth [KB/s]	N	C/N	Overhead
16	2	20	5.0%
16	5	20	4.5%
16	10	20	2.9%
16	20	20	3.3%
128	2	20	2.5%
128	5	20	3.3%
128	10	20	4.0%
128	20	20	4.7%
128	50	20	3.2%
128	100	20	2.9%
1024	2	20	2.9%
1024	5	20	3.3%
1024	10	20	3.2%
1024	20	20	2.6%
1024	50	20	3.6%
1024	100	20	4.4%
1024	200	20	4.6%

Table 5.2: Protocol overhead for different number of neighbours (N=neighbours, C/N=circuits per neighbour).

- FORWARD command headers (Table A.4)
- FLOW CONTROL commands (Table A.5)
- security-related data (Table 4.4)
- TCP/IP headers – overhead caused by TCP/IP headers is not included in our tests

Protocol overhead test was run with many different settings to see the influence of varying parameters. The Turtle turned out to be efficient as the overhead always fits between 2.5% and 5%.

Bandwidth [KB/s]	N	C/N	Overhead
16	20	1	3.4%
16	20	5	3.1%
16	20	20	3.3%
128	20	1	3.0%
128	20	5	3.1%
128	20	20	4.7%
128	20	100	4.2%
1024	20	1	2.8%
1024	20	5	3.2%
1024	20	20	2.6%
1024	20	100	4.7%
1024	20	500	3.7%

Table 5.3: Protocol overhead for different number of circuits per neighbour (N=neighbours, C/N=circuits per neighbour).

Bandwidth [KB/s]	N	C/N	Size	Overhead
16	5	20	2KB	4.5%
16	5	20	8KB	3.5%
16	5	20	32KB	4.2%
16	5	20	128KB	4.0%
128	25	20	2KB	3.8%
128	25	20	8KB	3.9%
128	25	20	32KB	4.7%
128	25	20	128KB	4.8%
1024	100	20	2KB	3.6%
1024	100	20	8KB	4.4%
1024	100	20	32KB	4.5%
1024	100	20	128KB	4.5%

Table 5.4: Protocol overhead for different sizes of communication buffer (N=neighbours, C/N=circuits per neighbour).

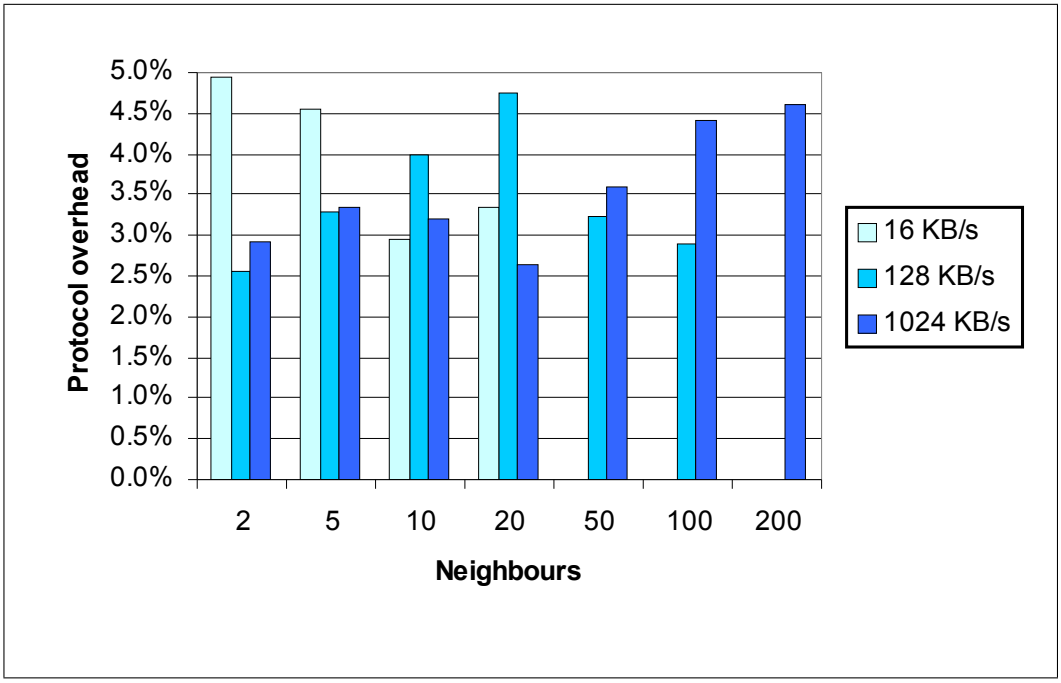


Figure 5.2: Protocol overhead for different number of neighbours.

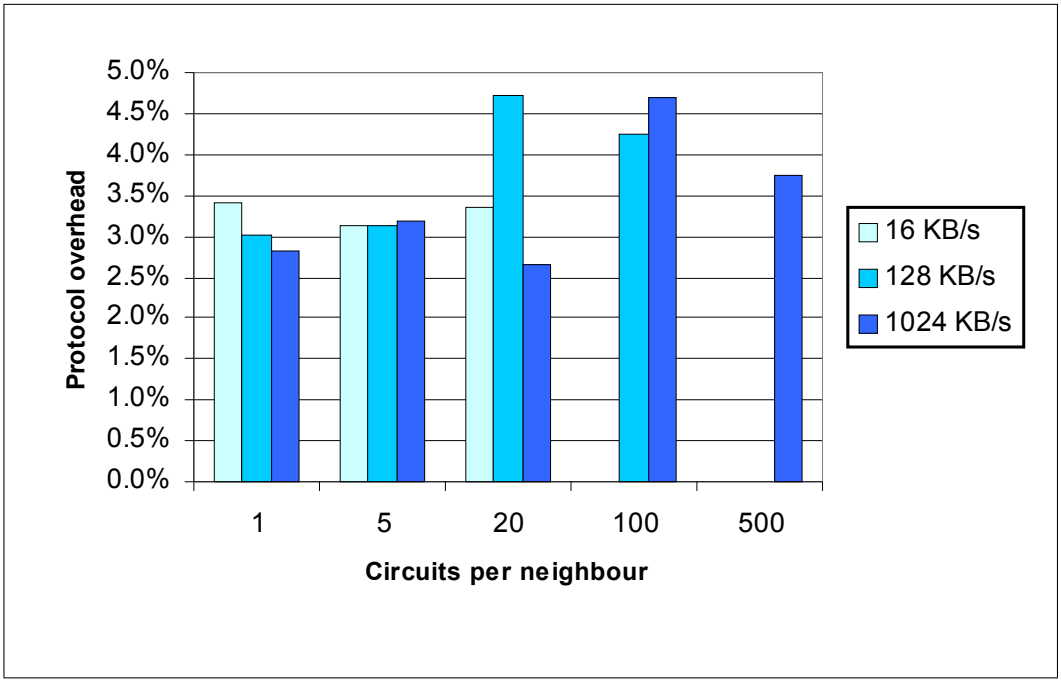


Figure 5.3: Protocol overhead for different number of circuits per neighbour.

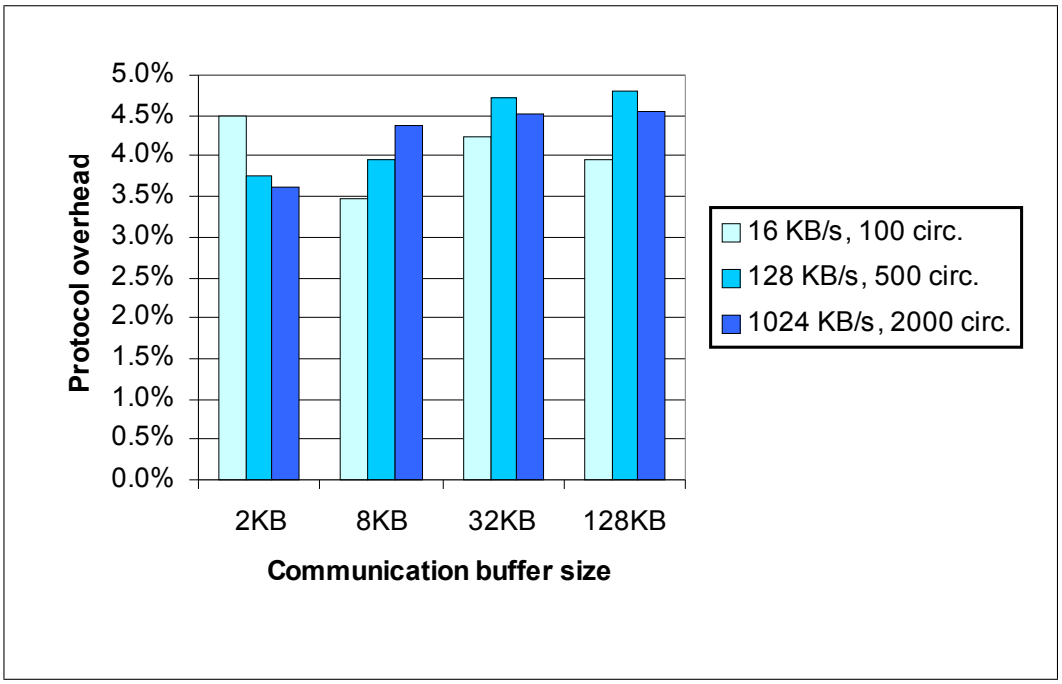


Figure 5.4: Protocol overhead for different sizes of communication buffer.

5.2 CPU load

In the second test CPU load of a Turtle node that relays a lot of traffic was measured. The experimental setup was exactly the same as in the first experiment – star topology with node 0 in the middle. Node 0 relayed all traffic going through the Turtle network. It did not share PC with any other node, so the process of node 0 was the only CPU consuming process running at that PC. Node 0 was also the only node with bandwidth limited by configuration. Other nodes' communication speed was limited only by node 0 – they sent and received as much data as node 0 allowed them to do.

CPU load measurement method was the same as in UNIX program *top* – jiffies spent in threads of Turtle were periodically recorded and divided by length of period (in jiffies of course). Measurement was done every two seconds for 15 minutes. Results were averaged from values recorded in the middle 6 minutes of the interval. The reasons for not using values at the beginning and at the end of the interval are same as in Protocol overhead tests (Section 5.1).

Because Turtle is a multi-threaded application, we had to measure all its threads. However, the only threads that participate in relaying traffic are the *router thread* and the *TCP thread*. Other threads cause negligible CPU load at relay node and were therefore omitted from this test. Total CPU

load was calculated as a sum of loads of router thread and CPU thread. Notice that total CPU load exceeds 100% in two tests – the reason is that each DAS2 PC has two processors, while CPU load is calculated per one processor (theoretically we could get 200% CPU load if both threads reached 100%).

CPU load test was run with many different settings to see the influence of varying parameters on router thread and TCP thread.

Bandwidth CPU load of both threads scales almost linearly with bandwidth. Total load of 100% is reached around 4 MB/s with security and 10 MB/s without security (there are separate limits on incoming and outgoing data, so 10 MB/s means 10 MB/s in and 10 MB/s out, not 5 MB/s in and 5 MB/s out).

Neighbours and Circuits per neighbour These parameters have negligible effect on CPU load, until total number of circuits gets extremely high. With 10000 circuits, the increase of CPU load is already significant (20% with 4 MB/s, see Table 5.7).

Security To analyze security overhead, we ran CPU load test with most CPU consuming security-related operations turned off. These operations are performed in TCP thread: encryption and decryption of data and computing and verifying of MACs (Message Authentication Codes). The security overhead turned out to be around 2/3 (2/3 of CPU cycles spent in security operations), except for runs with high number of circuits, when the security overhead decreased to almost 1/2.

		with security			without security		
N	**C/N**	**Router**	**TCP**	**Total**	**Router**	**TCP**	**Total**
5	5	0.9%	19.6%	20.5%	1.1%	5.7%	6.8%
5	10	1.0%	19.8%	20.8%	1.1%	5.3%	6.4%
5	50	0.8%	18.6%	19.4%	1.1%	4.4%	5.5%
5	100	1.2%	19.4%	20.6%	1.0%	4.1%	5.1%
10	5	0.9%	19.9%	20.8%	1.1%	5.1%	6.2%
10	10	0.9%	19.5%	20.4%	0.8%	4.4%	5.2%
10	50	1.3%	19.6%	20.9%	1.2%	4.6%	5.8%
10	100	1.9%	20.2%	22.1%	2.4%	5.6%	8.0%
50	5	1.1%	20.7%	21.8%	1.4%	5.0%	6.4%
50	10	1.2%	20.7%	21.9%	1.5%	4.7%	6.2%
50	50	3.3%	22.7%	26.0%	4.2%	7.8%	12.0%
50	100	2.4%	22.7%	25.1%	2.8%	7.1%	9.9%
100	5	1.2%	21.7%	22.9%	1.2%	5.9%	7.1%
100	10	2.2%	21.9%	24.1%	2.4%	6.2%	8.6%
100	50	2.2%	23.0%	25.2%	2.9%	7.6%	10.5%
100	100	2.3%	22.6%	24.9%	1.8%	6.0%	7.8%

Table 5.5: CPU load measurements of router thread, TCP thread and total load, 1 MB/s bandwidth (N=neighbours, C/N=circuits per neighbour).

		with security			without security		
N	**C/N**	**Router**	**TCP**	**Total**	**Router**	**TCP**	**Total**
5	5	2.0%	42.7%	44.7%	2.4%	10.8%	13.2%
5	10	1.7%	40.6%	42.3%	2.4%	10.5%	12.9%
5	50	1.4%	38.2%	39.6%	1.9%	9.6%	11.5%
5	100	1.4%	38.4%	39.8%	1.8%	8.4%	10.2%
10	5	1.8%	39.2%	41.0%	2.6%	11.4%	14.0%
10	10	1.8%	40.1%	41.9%	2.1%	9.5%	11.6%
10	50	1.8%	38.3%	40.1%	1.9%	8.6%	10.5%
10	100	2.6%	39.2%	41.8%	2.8%	9.7%	12.5%
50	5	1.6%	39.8%	41.4%	2.6%	10.6%	13.2%
50	10	1.6%	39.7%	41.3%	2.3%	8.9%	11.2%
50	50	4.9%	43.3%	48.2%	5.8%	12.7%	18.5%
50	100	6.0%	45.0%	51.0%	7.9%	15.5%	23.4%
100	5	2.2%	41.6%	43.8%	2.7%	10.8%	13.5%
100	10	2.6%	41.5%	44.1%	3.3%	10.2%	13.5%
100	50	6.5%	45.9%	52.4%	8.0%	15.8%	23.8%
100	100	4.9%	45.5%	50.4%	6.1%	13.9%	20.0%

Table 5.6: CPU load measurements of router thread, TCP thread and total load, 2 MB/s bandwidth (N=neighbours, C/N=circuits per neighbour).

		with security			without security		
N	**C/N**	**Router**	**TCP**	**Total**	**Router**	**TCP**	**Total**
5	5	3.5%	79.0%	82.5%	4.5%	22.2%	26.7%
5	10	3.5%	78.6%	82.1%	4.6%	22.3%	26.9%
5	50	3.0%	78.0%	81.0%	4.0%	19.2%	23.2%
5	100	3.1%	76.2%	79.3%	4.0%	18.9%	22.9%
10	5	3.5%	79.0%	82.5%	4.4%	22.7%	27.1%
10	10	3.6%	79.6%	83.2%	4.5%	21.0%	25.5%
10	50	3.0%	75.8%	78.8%	4.0%	18.4%	22.4%
10	100	3.1%	75.9%	79.0%	3.5%	19.4%	22.9%
50	5	3.6%	80.3%	83.9%	4.6%	26.0%	30.6%
50	10	3.5%	79.0%	82.5%	4.4%	23.6%	28.0%
50	50	6.0%	80.7%	86.7%	6.5%	23.3%	29.8%
50	100	10.2%	85.2%	95.4%	10.0%	26.7%	36.7%
100	5	3.6%	81.0%	84.6%	4.3%	23.8%	28.1%
100	10	3.9%	80.3%	84.2%	4.9%	21.6%	26.5%
100	50	10.5%	87.8%	98.3%	13.0%	26.7%	39.7%
100	100	13.3%	91.9%	105,2%	15.6%	31.9%	47.5%

Table 5.7: CPU load measurements of router thread, TCP thread and total load, 4 MB/s bandwidth (N=neighbours, C/N=circuits per neighbour).

		with security			without security		
N	**C/N**	**Router**	**TCP**	**Total**	**Router**	**TCP**	**Total**
5	5	N/A	N/A	N/A	5.9%	50.8%	56.7%
5	10	N/A	N/A	N/A	10.8%	55.9%	66.7%
5	50	N/A	N/A	N/A	10.3%	52.7%	63.0%
5	100	N/A	N/A	N/A	10.3%	48.8%	59.1%
10	5	N/A	N/A	N/A	11.7%	56.6%	68.3%
10	10	N/A	N/A	N/A	10.4%	54.1%	64.5%
10	50	N/A	N/A	N/A	10.5%	51.0%	61.5%
10	100	N/A	N/A	N/A	10.1%	52.3%	62.4%
50	5	N/A	N/A	N/A	12.7%	72.0%	84.7%
50	10	N/A	N/A	N/A	11.7%	66.3%	78.0%
50	50	N/A	N/A	N/A	11.0%	61.9%	72.9%
50	100	N/A	N/A	N/A	18.2%	66.1%	84.3%
100	5	N/A	N/A	N/A	12.3%	71.4%	83.7%
100	10	N/A	N/A	N/A	12.4%	69.2%	81.6%
100	50	N/A	N/A	N/A	16.8%	73.1%	89.9%
100	100	N/A	N/A	N/A	28.8%	81.1%	109,9%

Table 5.8: CPU load measurements of router thread, TCP thread and total load, 10 MB/s bandwidth (N=neighbours, C/N=circuits per neighbour).

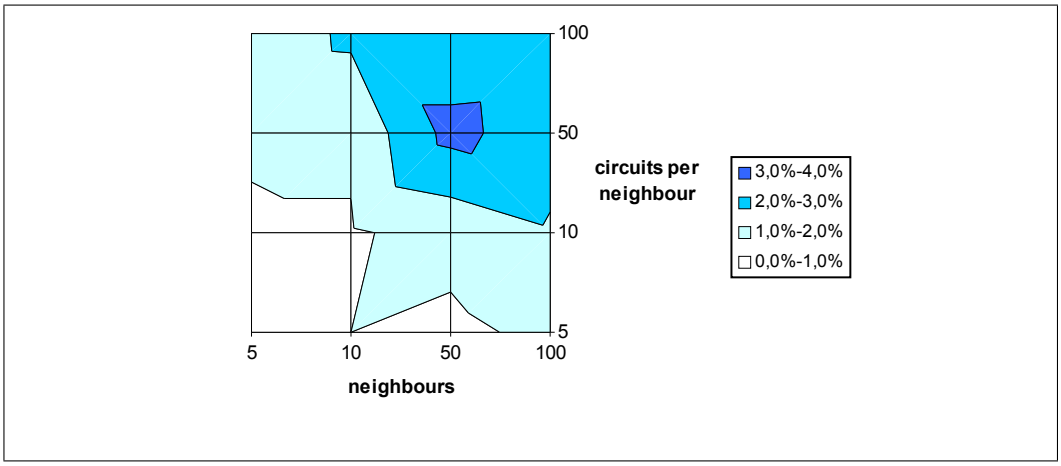


Figure 5.5: Router thread CPU load, with security, 1 MB/s bandwidth

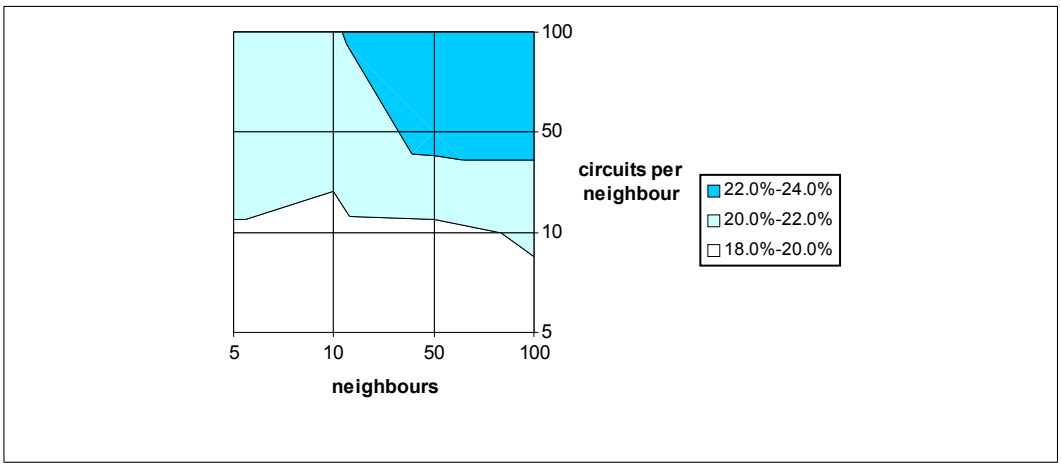


Figure 5.6: TCP thread CPU load, with security, 1 MB/s bandwidth

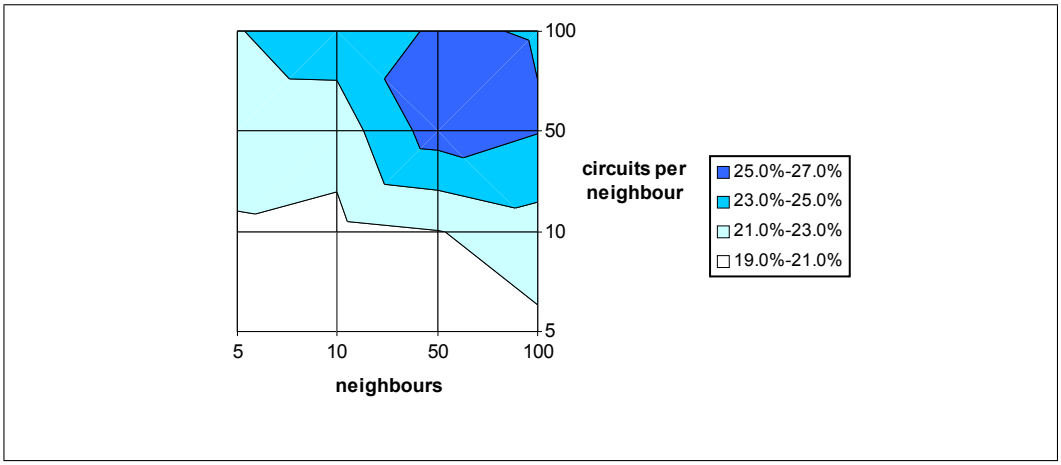


Figure 5.7: Total CPU load, with security, 1 MB/s bandwidth

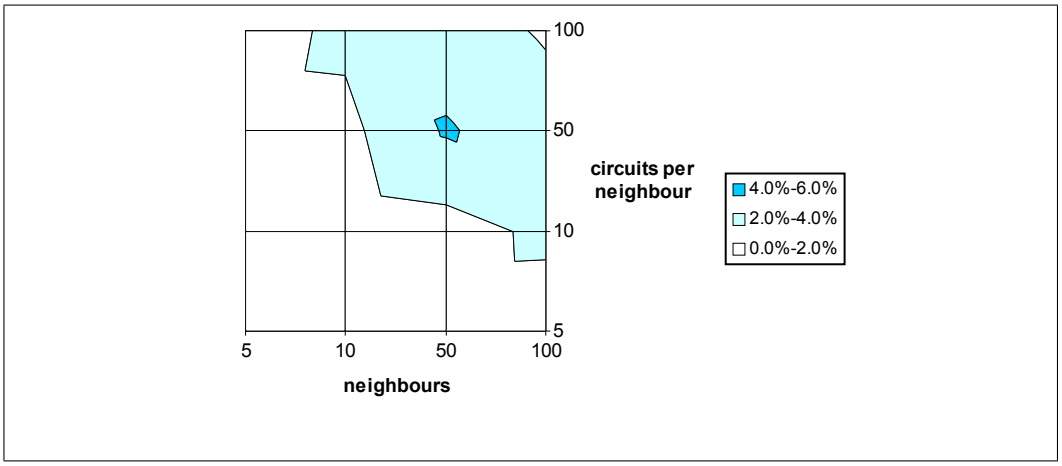


Figure 5.8: Router thread CPU load, without security, 1 MB/s bandwidth

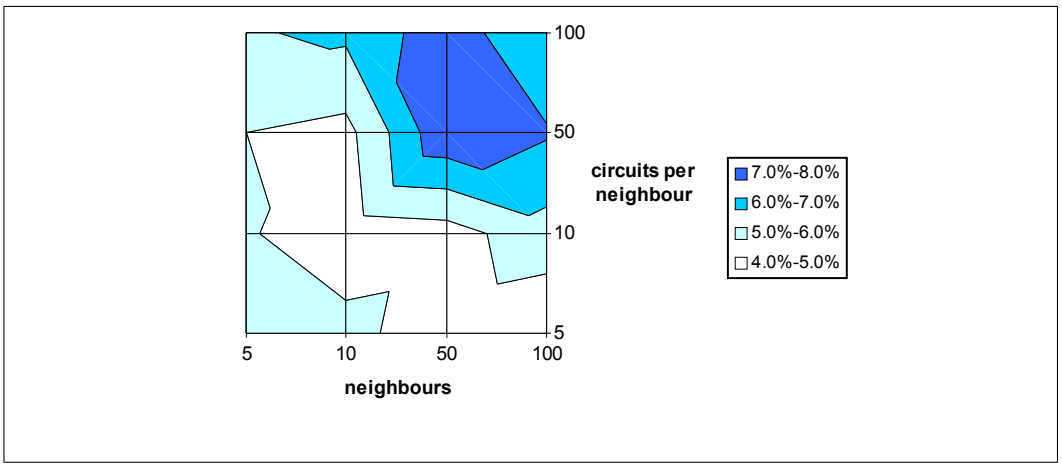


Figure 5.9: TCP thread CPU load, without security, 1 MB/s bandwidth

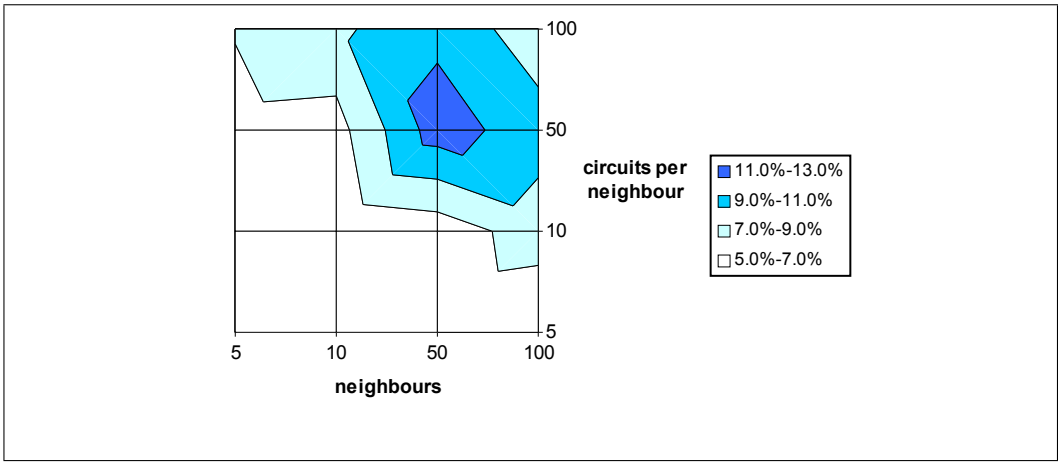


Figure 5.10: Total CPU load, without security, 1 MB/s bandwidth

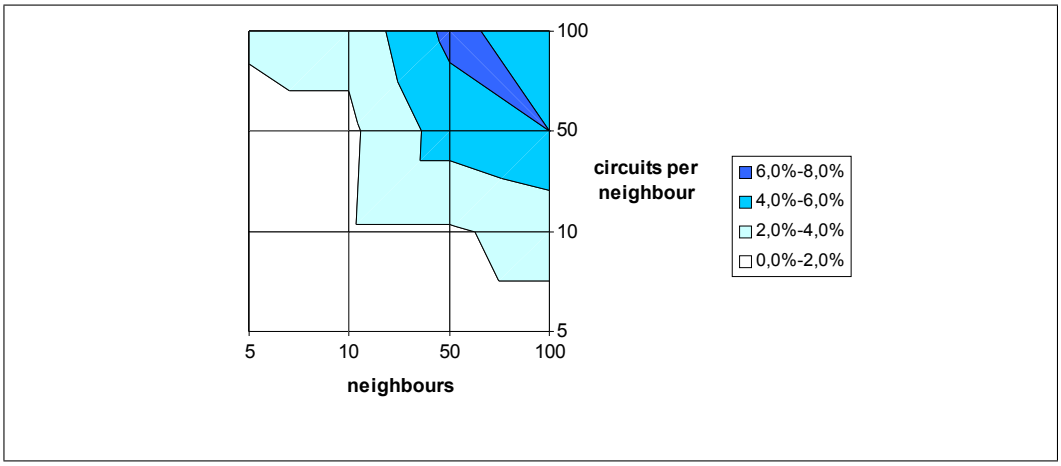


Figure 5.11: Router thread CPU load, with security, 2 MB/s bandwidth

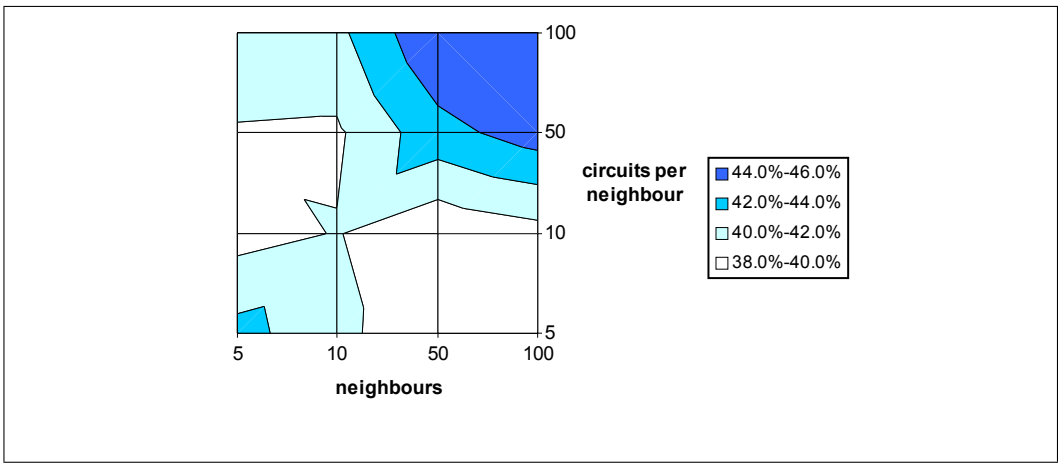


Figure 5.12: TCP thread CPU load, with security, 2 MB/s bandwidth

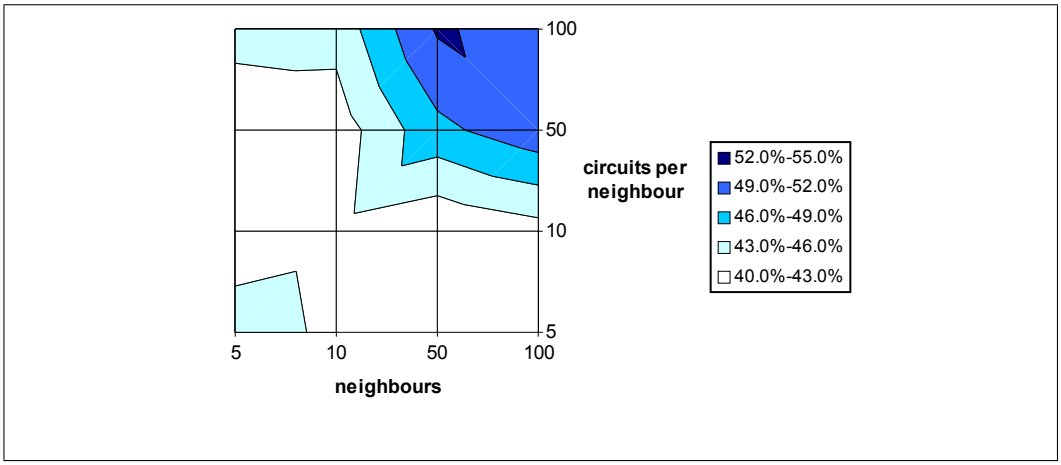


Figure 5.13: Total CPU load, with security, 2 MB/s bandwidth

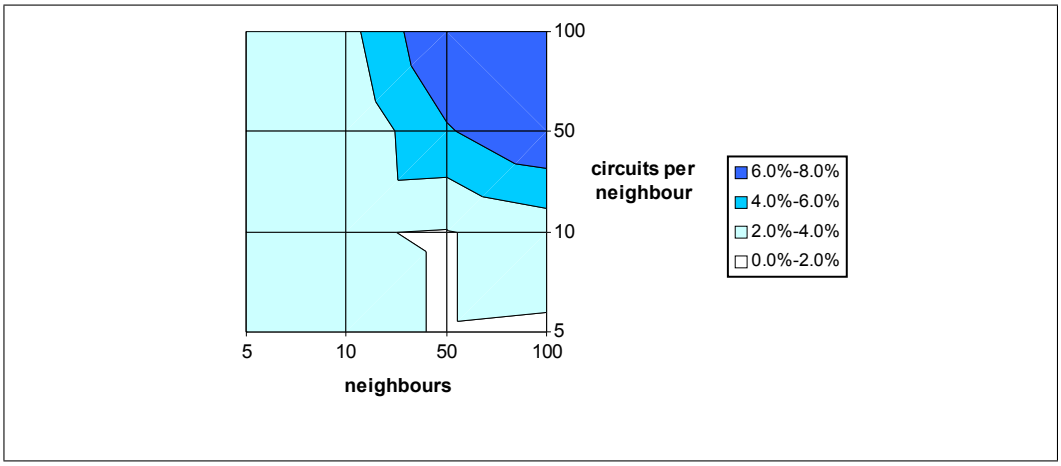


Figure 5.14: Router thread CPU load, without security, 2 MB/s bandwidth

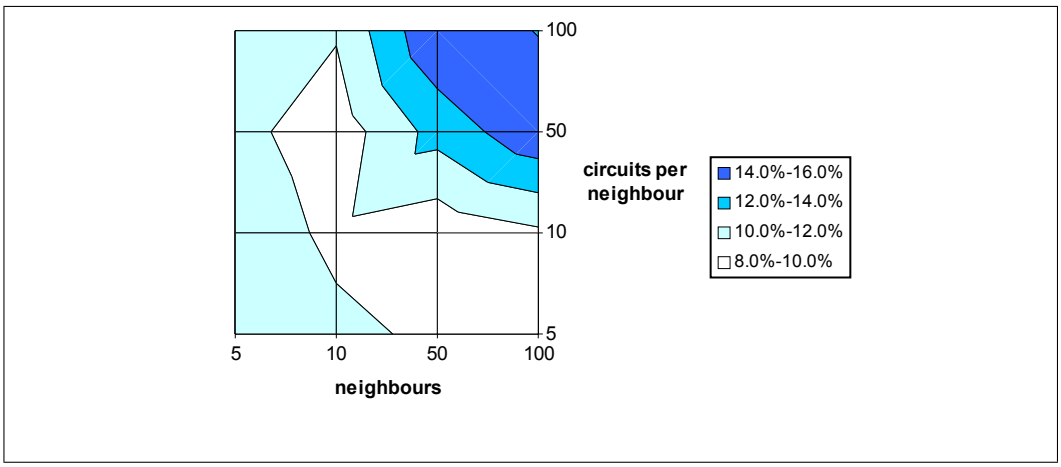


Figure 5.15: TCP thread CPU load, without security, 2 MB/s bandwidth

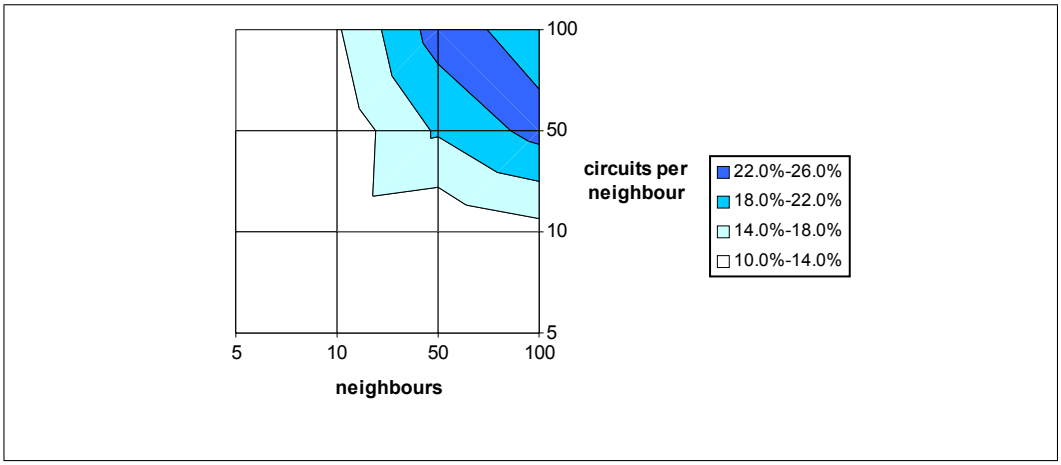


Figure 5.16: Total CPU load, without security, 2 MB/s bandwidth

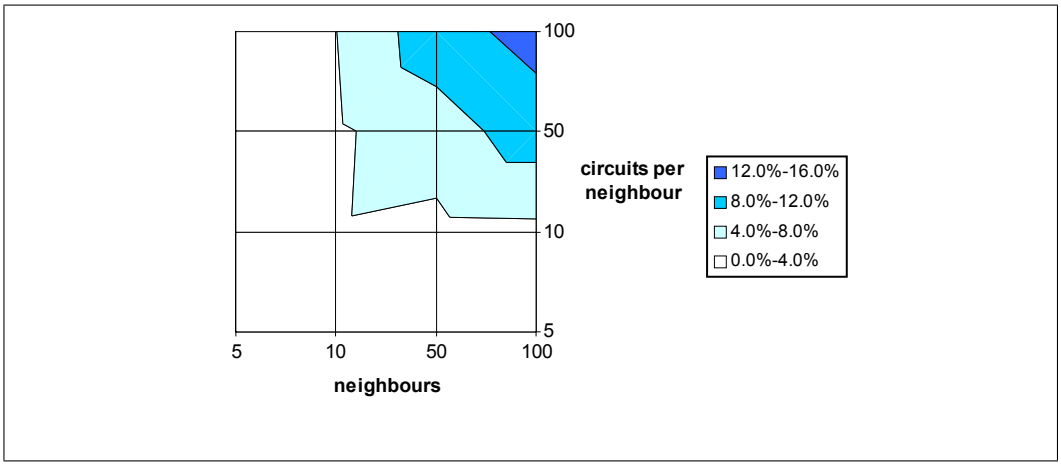


Figure 5.17: Router thread CPU load, with security, 4 MB/s bandwidth

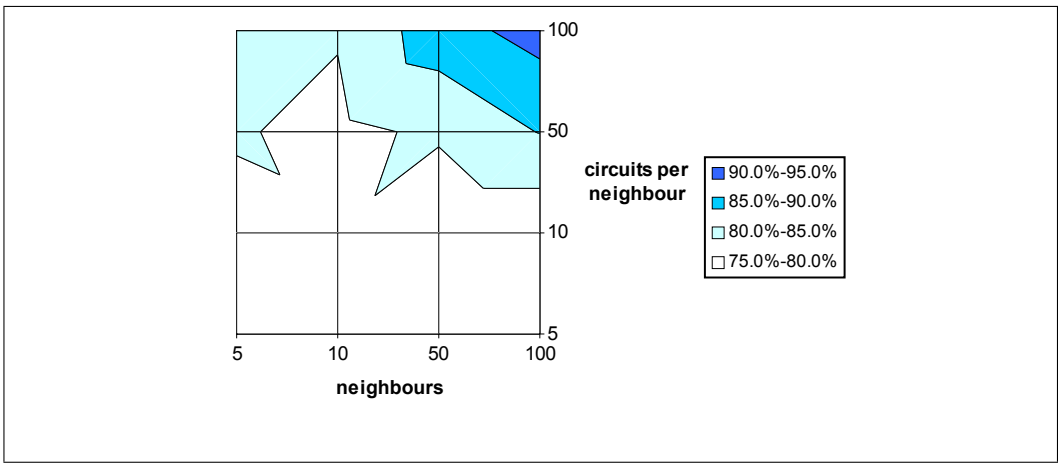


Figure 5.18: TCP thread CPU load, with security, 4 MB/s bandwidth

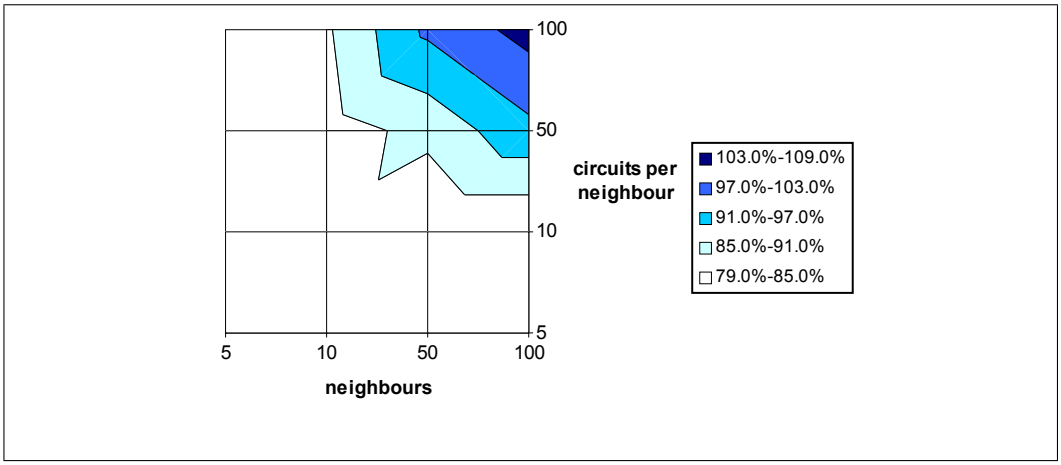


Figure 5.19: Total CPU load, with security, 4 MB/s bandwidth

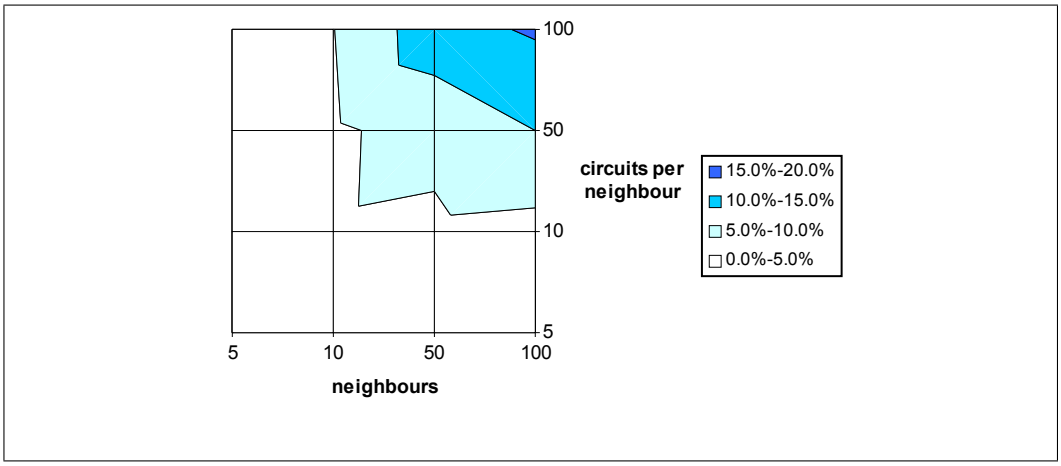


Figure 5.20: Router thread CPU load, without security, 4 MB/s bandwidth

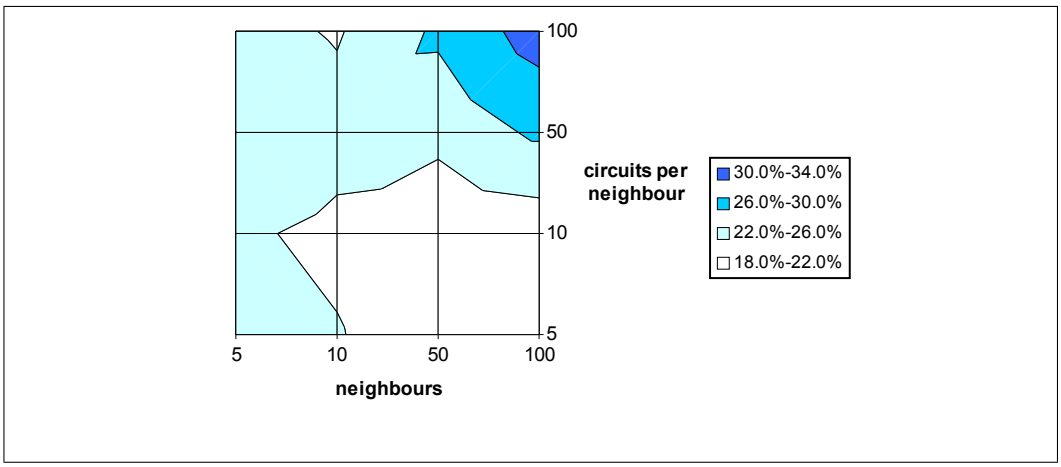


Figure 5.21: TCP thread CPU load, without security, 4 MB/s bandwidth

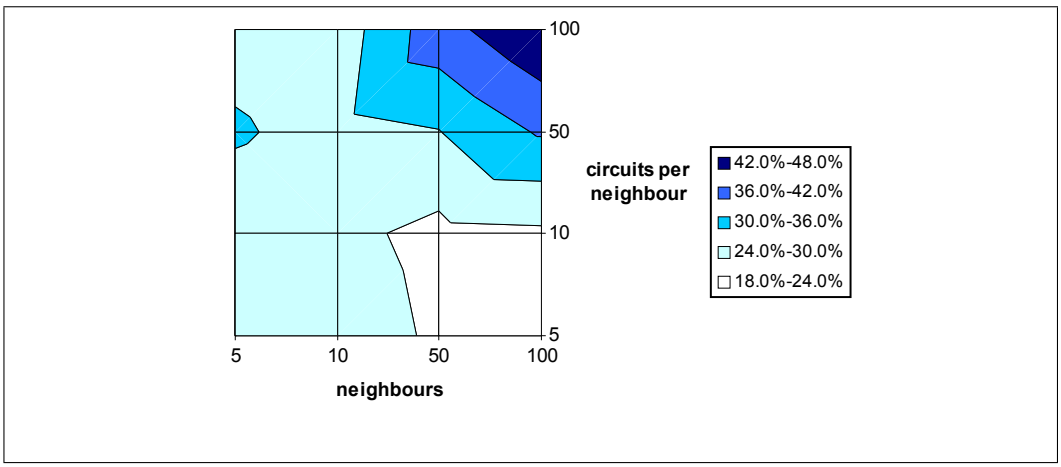


Figure 5.22: Total CPU load, without security, 4 MB/s bandwidth

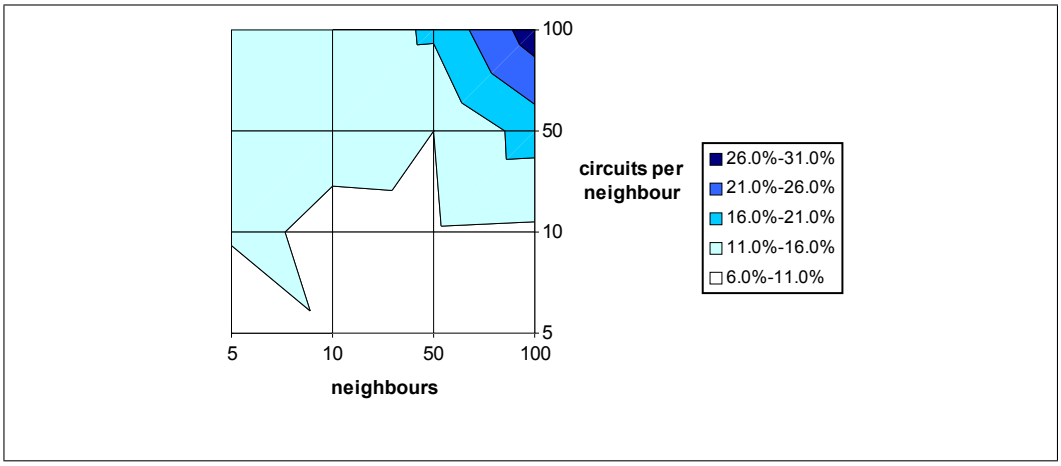


Figure 5.23: Router thread CPU load, without security, 10 MB/s bandwidth

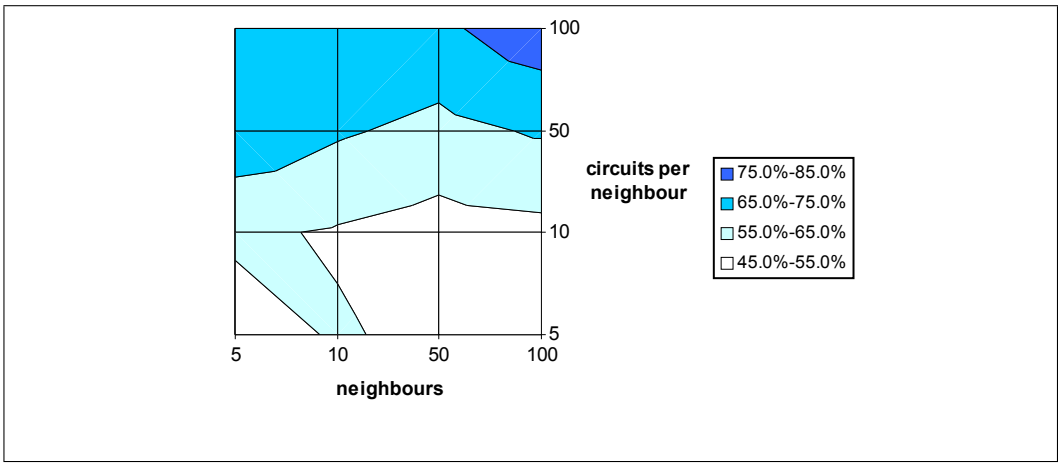


Figure 5.24: TCP thread CPU load, without security, 10 MB/s bandwidth

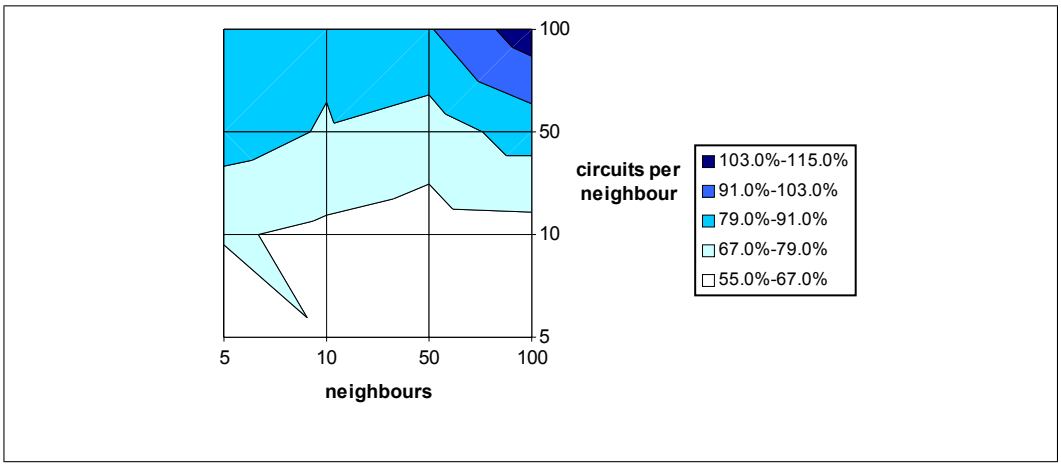


Figure 5.25: Total CPU load, without security, 10 MB/s bandwidth

5.3 Latency

In the last test we measured latency of data transferred through a single segment virtual circuit. This time the Turtle network consisted of two nodes – node 0 was a data source that sent file to node 1. Both nodes ran on the same PC, so the measurement was not influenced by network latency and there was no need to synchronize system clock of two different PCs. Bandwidth of both nodes was limited by configuration.

The test startup procedure included several steps. First, nodes 0 and 1 established TCP connection to each other. Then node 0 issued a query to find the file at node 1. After receiving reply, node 0 opened certain number of file download circuits and started downloading the file on all of them. One of the circuits was selected as a "measurement circuit" – latency was calculated as a difference between time, when file data left File Sender at node 1, and time of arrival of file data to File Receiver at node 0. Measurements were repeated until there were enough values collected (50 in most cases, 10 in cases with very high latency).

Table 5.9 shows latency test results obtained with different bandwidth settings and different number of circuits. The results look surprisingly bad, but in fact they perfectly meet our expectations. To explain why, let us look how file data travel from node 1 to node 0. The data start their journey in send buffer of File Sender at node 1 (recall Figure 4.9). To minimize effect of this buffer, we disabled it, so the data go directly to buffer of Communication Channel. Here, data of all circuits are multiplexed to TCP connection to node 0 (recall Figure 4.3). After arriving to node 0, file data are propagated through Communication Channel and Router directly to File Receiver.

Because bandwidth of the connection between nodes 0 and 1 is limited, Communication Channel at node 1 cannot send data as quickly as it receives them from File Sender, and buffers of all circuits get full very soon after transfers begin. They stay full during the whole test, because whenever Communication Channel sends some data to node 0, it also sends FLOW CONTROL command to File Sender, which immediately fills buffer with more data. Data latency measurement therefore begins when data are sent to nearly full buffer of Communication Channel and ends soon after they leave the buffer (transfer time between node 0 and 1 is negligible).

Because data spent most of the time just waiting in the buffer of Communication Channel, we can calculate expected latency as $BUF/(BW * E/C)$, where BUF is size of the buffer (32KB in our case), BW is total bandwidth, E is effectiveness of the protocol (0.96 in our case, see Section 5.1) and C is number of circuits. Latency tests show that this calculation is correct, although measured values are usually a bit lower than expected values. There

are two reasons for this phenomenon. First, buffers in Communication Channel are not always full, because FLOW CONTROL commands are sent only if space in the buffer is bigger than certain threshold (to prevent sending too many commands). And second, due to bandwidth limiting mechanism, data are usually communicated only at the beginning of each second, until bandwidth limit for that second is reached. Then the communication is blocked, until next second starts. This behaviour influences especially measurements with low expected latencies (< 1 second).

Despite very high latencies, Turtle can still serve its purpose very well. For long continuous transfers, latency is usually not as important as throughput. For short transfers, latency will not be so high, because buffers in Communication Channel will be empty. Moreover, implementation of data multiplexor in Communication Channel prioritizes circuits that do not send much data, which makes interactive communication even faster.

Bandwidth [KB/s]	Circuits	Latency [s]	Expected latency [s]
2	1	18.50	16.67
2	2	33.70	33.33
2	5	84.50	83.33
2	10	167.70	166.67
16	1	2.00	2.08
16	2	4.20	4.17
16	5	10.20	10.42
16	10	20.20	20.83
16	20	40.49	41.67
16	50	102.00	104.17
16	100	204.00	208.33
128	1	0.00	0.26
128	2	0.01	0.52
128	5	1.00	1.30
128	10	2.00	2.60
128	20	4.60	5.21
128	50	11.40	13.02
128	100	24.59	26.04
128	200	50.19	52.08
1024	1	0.00	0.03
1024	2	0.00	0.07
1024	5	0.01	0.16
1024	10	0.03	0.33
1024	20	0.15	0.65
1024	50	1.14	1.63
1024	100	2.99	3.26
1024	200	5.80	6.51
1024	500	15.23	16.28

Table 5.9: Data latency measurements.

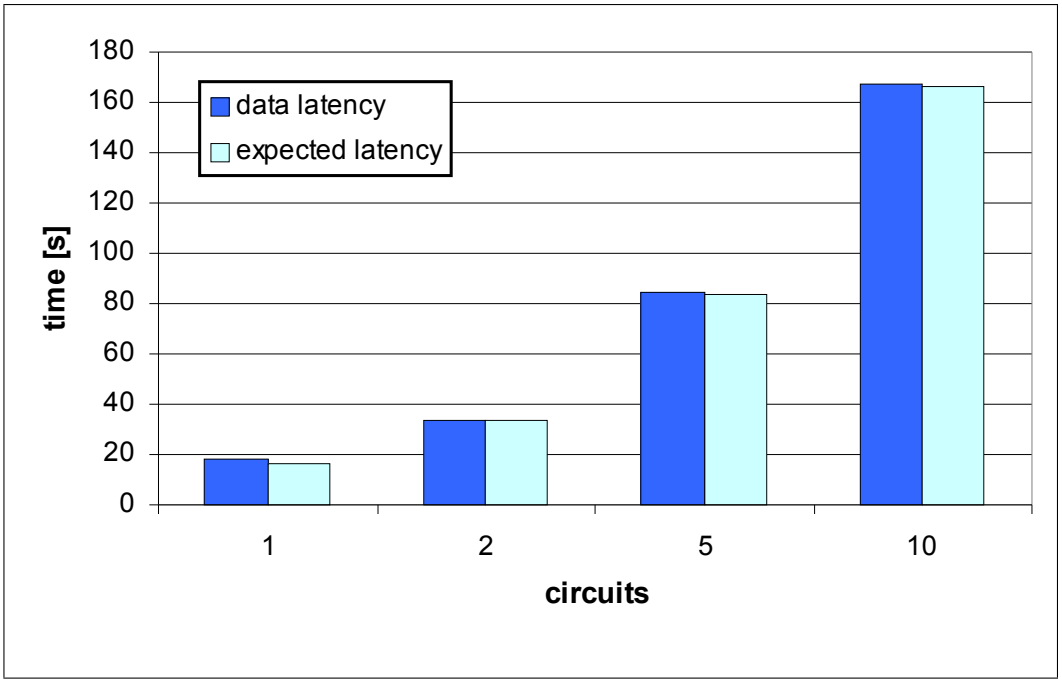


Figure 5.26: Data latency, 2 KB/s bandwidth

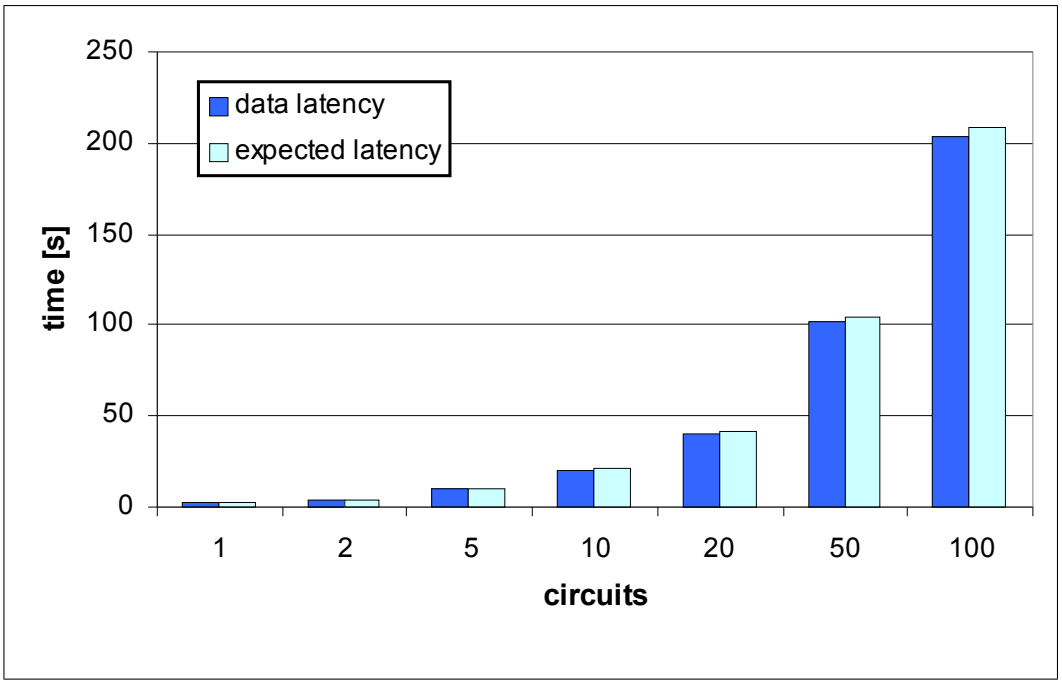


Figure 5.27: Data latency, 16 KB/s bandwidth

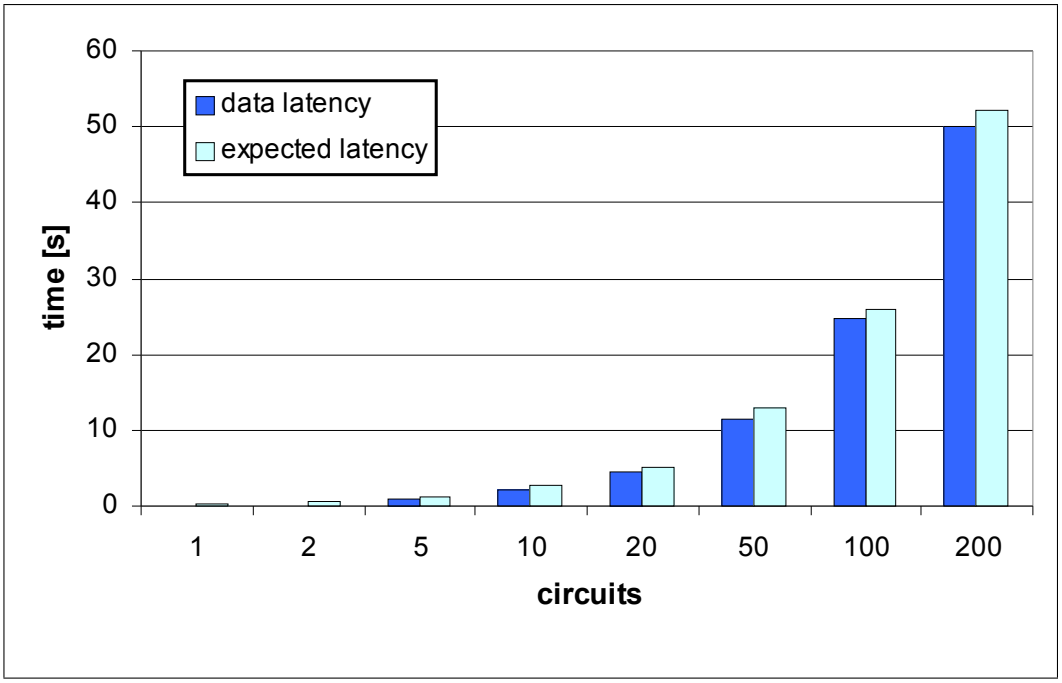


Figure 5.28: Data latency, 128 KB/s bandwidth

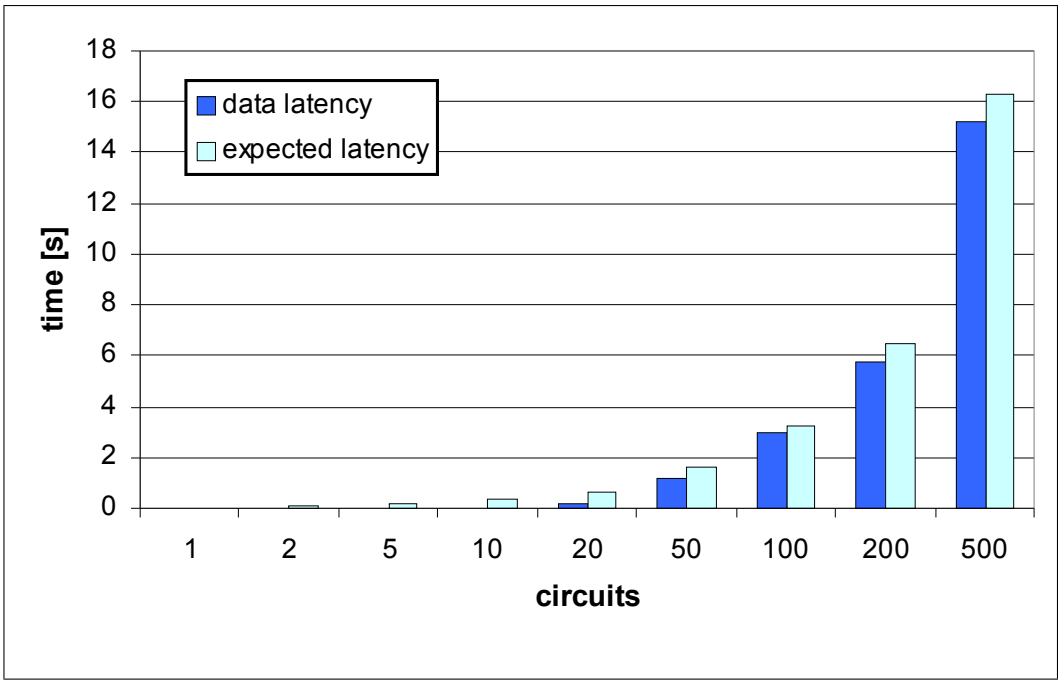


Figure 5.29: Data latency, 1 MB/s bandwidth

Chapter 6

Orkut data analysis

During the design of Turtle we made several assumptions about how the Turtle network will look, when many people start using it. Verifying our assumptions is difficult, but it is important if we want to be sure that the network will not collapse. The main questions we asked ourselves were:

- Do Turtle nodes form a fully connected graph?
- What if $x\%$ of nodes are down? Does it cause the network to split into separate components?
- How many neighbours do Turtle nodes usually have?
- What is the average/maximum length of path in the network? How is it influenced by shutting down $x\%$ of nodes?

Giving precise answers to these questions is not possible at present. First, there is no Turtle network running, which we could analyse. And second, the network architecture prevents anyone from doing deeper analysis. Due to these facts we decided to take a different approach and we analysed data from Orkut[25] web site.

Orkut is an online community web site ran by Google. After registration at Orkut, user is asked to enter tons of personal information. Orkut then provides various services to its users, which we will not describe here. For us, it is important that each user maintains a list of friends selected from other Orkut users. The list, together with personal information, is visible to all Orkut users, which makes crawling the whole web technically possible (only by people with account at Orkut, of course). For our analysis, we did not need any personal data about users. We needed just the graph, where each node denotes a user and edge denotes a relationship. Thanks to Rolan Yang,

who collected Orkut data for his GeOrkut project[33], we obtained the data for our analysis.

The web site was crawled at the beginning of year 2004, when Orkut had much less users than it has now. The graph contains 106481 nodes and 475800 undirected edges. The graph is fully connected, because users are allowed to register only after invitation from already registered user. New user therefore gets a connection to the main graph component during the registration process.

We believe that the graph obtained from Orkut is similar to graph that would be created by Turtle users. In both systems, virtual relationships are based on real life friendships. This is true for most Orkut users, although there are some who have suspiciously high number of friends. In Turtle, where relationships require higher level of trust than in Orkut, such users would not exist – it would be too risky. Despite this difference, Orkut graph is probably the best approximation of what we would get with Turtle.

6.1 Connectivity and components

In this section, we try to answer questions related to node connectivity (i.e. how many neighbours nodes have) and graph components. Distribution of node connectivity is shown in Figure 6.1. There are two interesting points about this figure:

- 30.3% of nodes have only one neighbour. Each such node is totally dependent on its neighbour and if the neighbour is shut down, the node gets disconnected from the network. We see two possible reasons for not having more than one neighbour: First, the user does not know anyone else at Orkut and none of his friends is able to or interested in joining it. And second, user just wanted to try Orkut and is not interested in using it and finding or inviting new friends. Although there is no evidence for it, the second reason seems more likely. If it is true and if the behaviour of Turtle users is similar, nodes with one neighbour are not as big problem as it looks, because they will not be very active.

- 8.8% of nodes have 25 neighbours or more. Having so many friends could be considered risky in Turtle, where each relationship requires certain level of trust.

Figure 6.2 shows changing distribution of node connectivity when randomly selected nodes are shut down. The fraction of nodes, which are shut

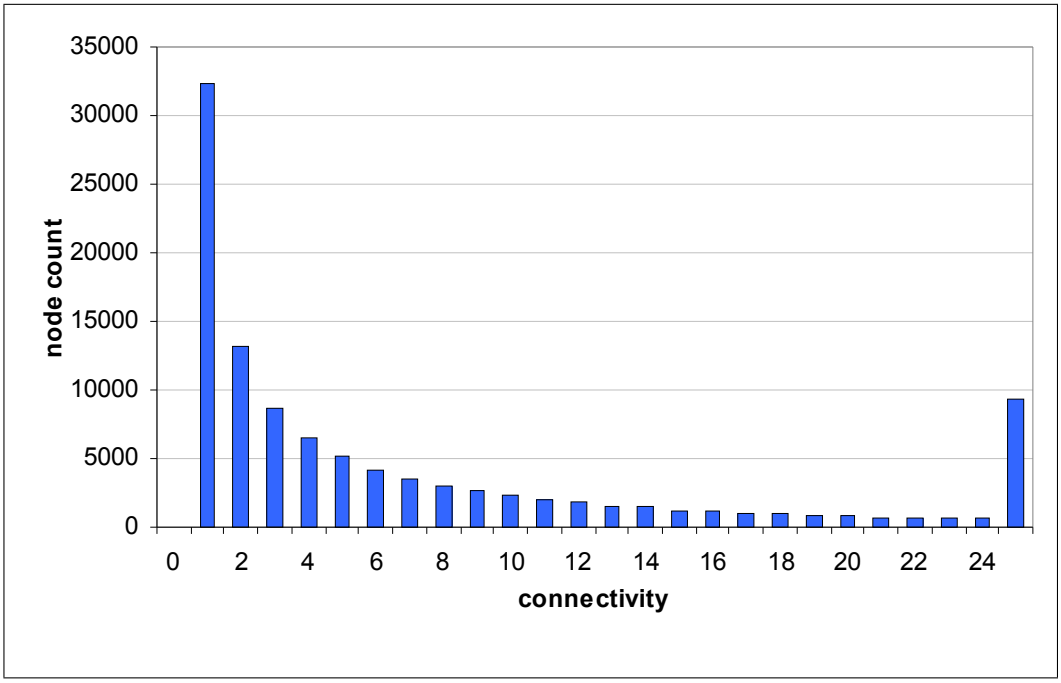


Figure 6.1: Distribution of node connectivity in the full graph

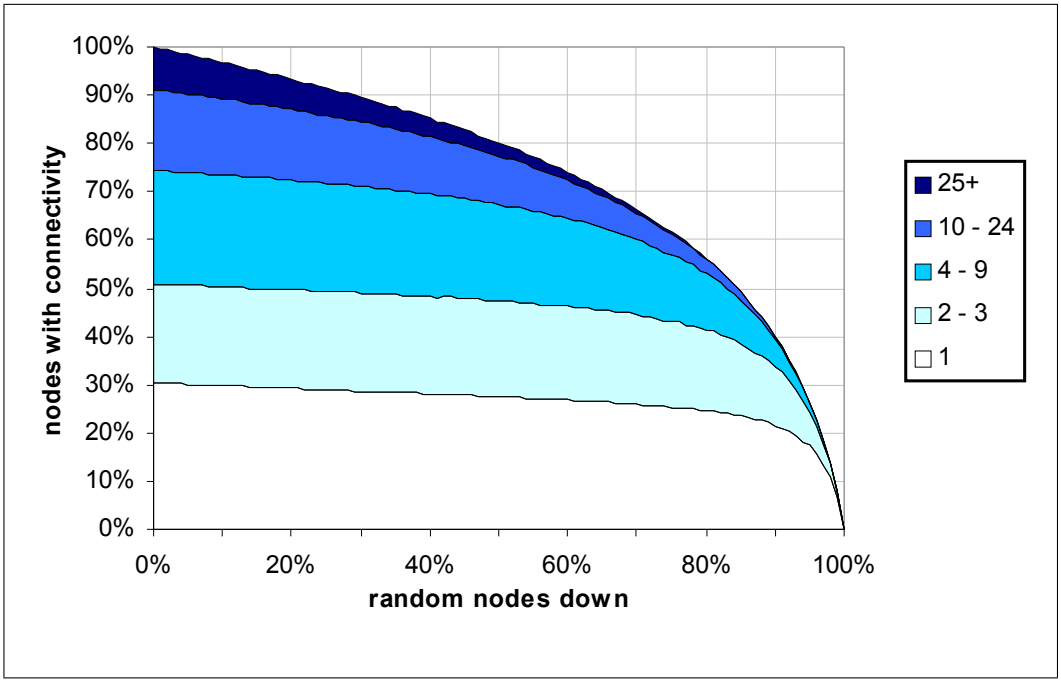


Figure 6.2: Distribution of node connectivity

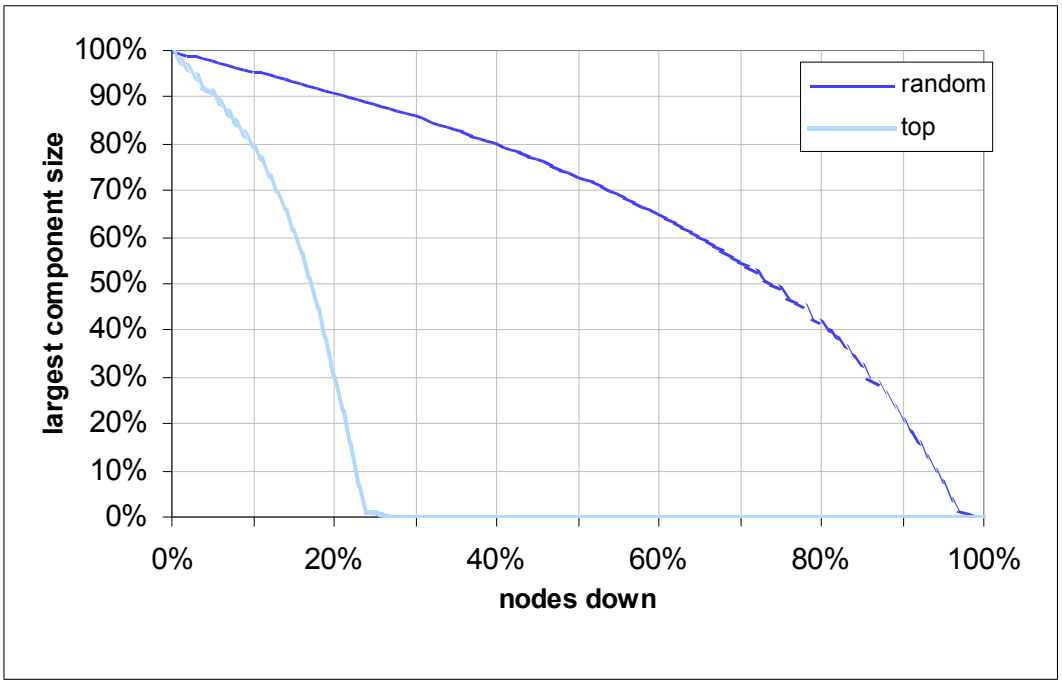


Figure 6.3: Size of the largest component (100% = all nodes)

down, increases smoothly from 0% (full graph) to 100% (all nodes down)[1]. The figure shows that number of high connectivity nodes goes down first. Number of low connectivity nodes decreases slowly at the beginning, because high connectivity nodes are becoming lower connectivity nodes. On the other hand, zero connectivity increase from left to right. Slowly at the beginning (when 50% nodes are down, 20% of the remaining nodes have no neighbour) and quickly in the right part of the figure.

Another property of Orkut graph is the size of the largest component. It is shown in Figure 6.3, again for changing fraction of random nodes down and the most connected nodes down. Size of the largest component is important property of the network. It tells us, whether the network splits into multiple separate networks if certain fraction of nodes is shut down. When random nodes are shut down, the network seems to be quite resistant. With 50% randomly selected nodes down, the largest component contains 73.2% of the remaining nodes. This is very optimistic number, if we consider that more than a half of the disconnected nodes have only one neighbour in the full graph (i.e. they know they can be easily disconnected).

[1]For each percentage the test was repeated 10 times and the results were averaged. This method was applied to the all tests in this chapter, where random numbers/random selection was applied.

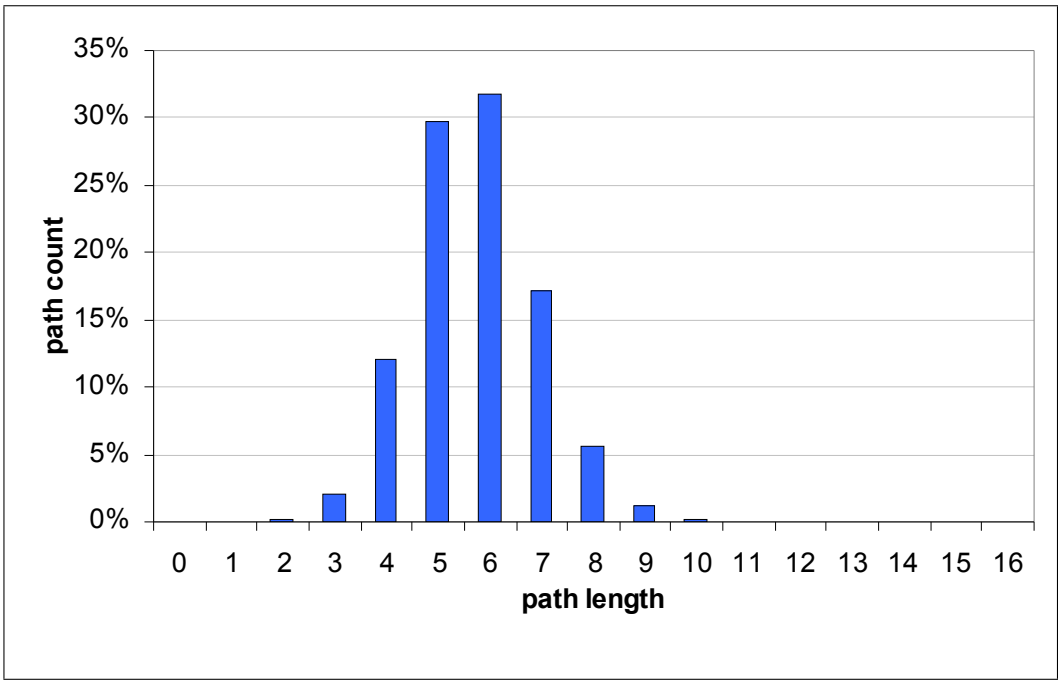


Figure 6.4: Distribution of path lengths in the full graph

The situation is completely different, when nodes are shut down from the most connected to the least connected. The size of the largest component drops below 1% when 25% of the most connected nodes are down. The network could be attacked in this way, but it would require an extremely powerful adversary. On the other hand, notice that removing only 8.8% of the most connected nodes (those risky ones with too many neighbours) does not harm the network too much.

There are few more interesting statistics, we have acquired from the Orkut graph. The largest biconnected component contains 64.5% of nodes. No node in this component can be disconnected from it by removing any other single node. From the remaining 35.5% nodes, most have one neighbour (30.3% of all nodes), which is the dangerous one. 5.2% of nodes have at least two neighbours, but still they can be disconnected from the main component by removing single node. The largest subgraph, which is connected to the main component via one node, contains 23 nodes.

6.2 Data availability

The second group of measurements focuses on data availability and data distance in the Orkut graph. Figure 6.4 shows distribution of path lengths in

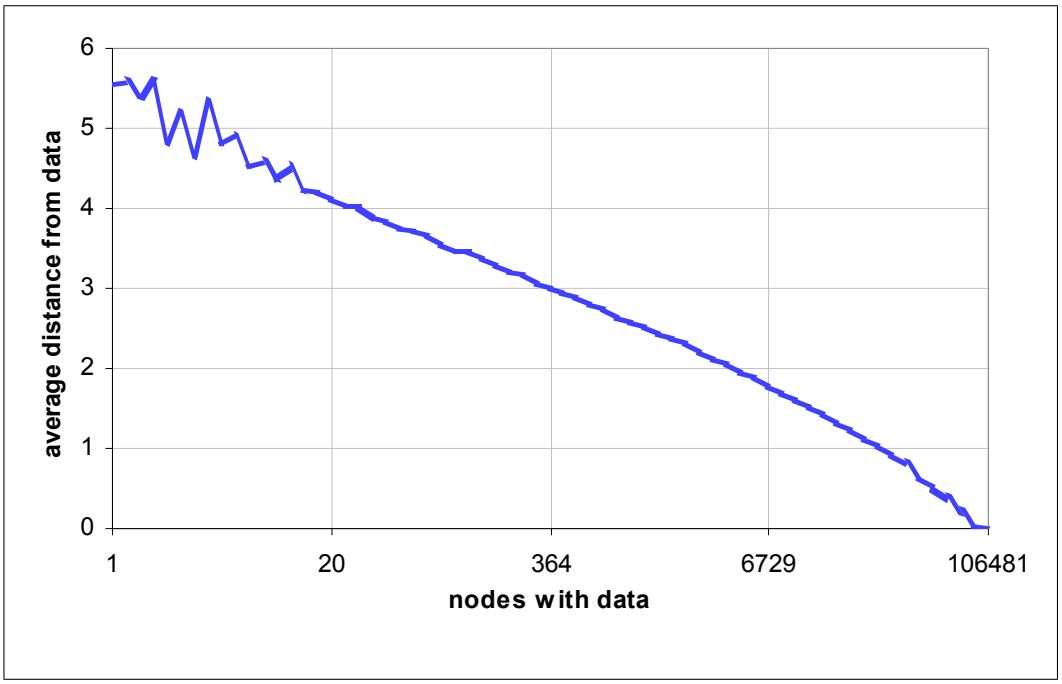


Figure 6.5: Average distance from data in the full graph. Horizontal axis has logarithmic scale

the full graph. The distribution is very similar to normal distribution with mean 5.72 and standard deviation 1.21. The longest path has 16 hops.

The average path length says, what is the expected distance from random node to another random node. In the real network, however, downloads are established in different way. If there are more hits for a query, user usually chooses the hit that comes first, i.e. hit that has the lowest distance from the user's node. Figure 6.5 takes this behaviour into account. It shows average distance from data, if there are more sources of the desired data. The number of data sources varies from one node to all nodes. The distance decreases from average path length to zero.

The following measurements were performed on a graph with 10, 100 and 1000 data sources. All three variants give similar results, so we show only results of 100 data source variant. The results were collected on graphs with randomly selected nodes shut down and the most connected nodes shut down. Curves in Figure 6.6 represent decreasing data availability in graphs with increasing number of nodes down. The curves are almost identical to curves that represent the largest component size (Figure 6.3). The reason is that the largest component almost always contains at least one data source and very small fraction of other components contains data sources. In other words, having connection to the largest component is more or less equivalent

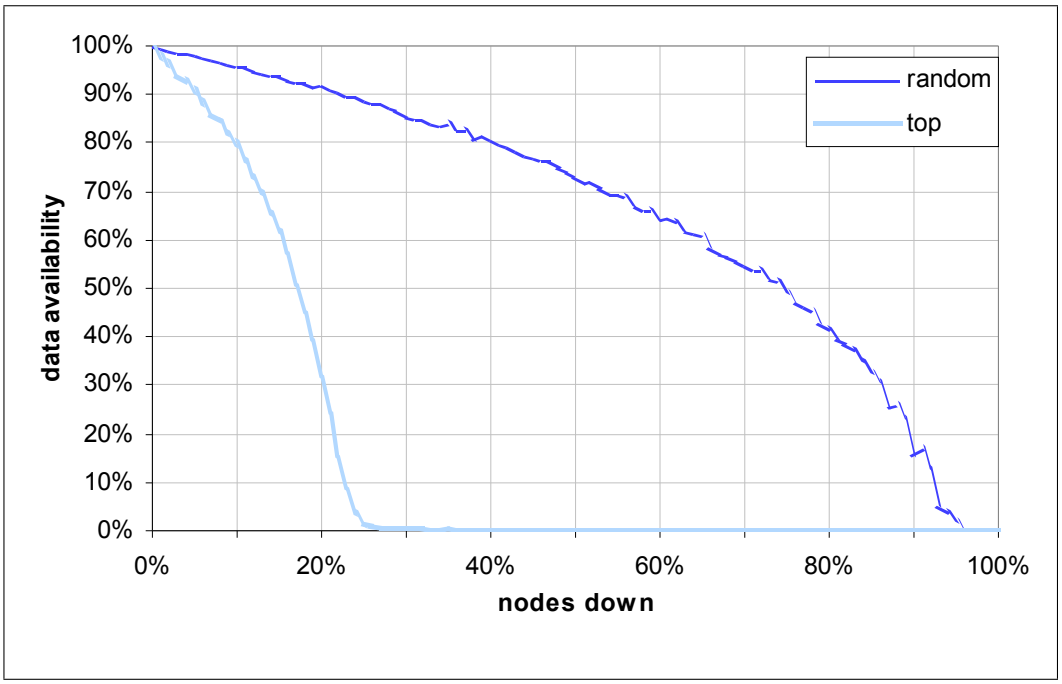


Figure 6.6: Data availability in graph with 100 data sources

to having (probably indirect) connection to some data source.

Figures 6.7 and 6.8 show average and maximum distance from data in graph with 100 data sources. When random nodes are shut down, both distances slowly increases until 90% of nodes are down. Then the distances drop down quickly, because components with data sources become small. Different situation appears when the most connected nodes are shut down. The distances then increase until 23% of nodes (i.e. all nodes with 12 neighbours or more and half of nodes with 11 neighbours) are down. At this point, the distances become extremely high – maximum path length in such graph exceeds 150! We have not done detailed analysis of this graph, so we do not know reasons for such extreme values. What we know is that after 23% of nodes being down, the graph splits into many small components and the average and maximum distance from data decreases drastically.

The results of our measurements make us believe that Turtle could successfully operate in the real life. The volume of data transferred over the network should not be higher than 10 times more than it would take in normal peer-to-peer network, where data travel directly between endpoints. The network seems to be resistant to removing few percent of highly connected nodes or removing lots of random nodes. Only nodes with single neighbour are in danger of being easily disconnected from the network.

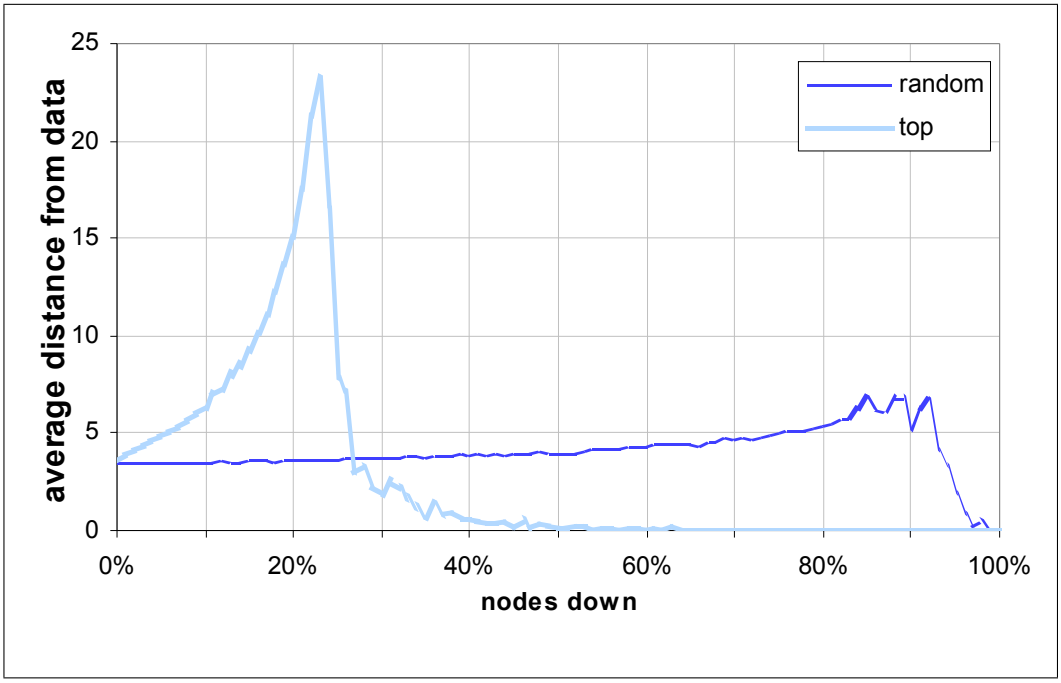


Figure 6.7: Average distance from data in graph with 100 data sources

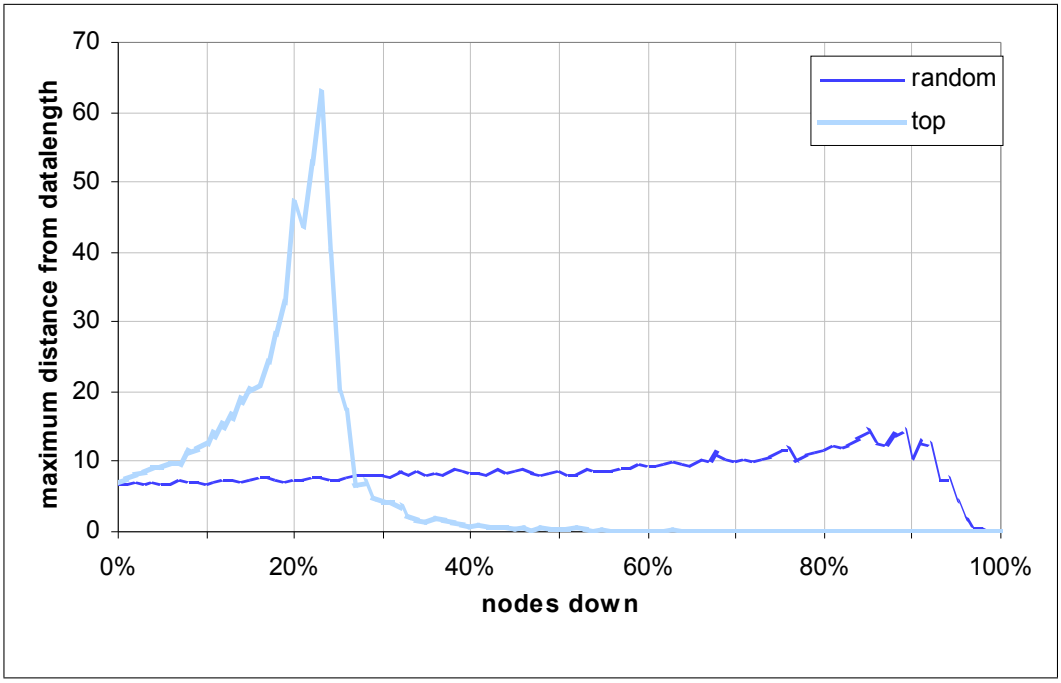


Figure 6.8: Maximum distance from data in graph with 100 data sources

Chapter 7

Conclusions and future work

In this thesis we have described Turtle, a peer-to-peer architecture for safe sharing of sensitive data. In order to achieve strong privacy guarantees, Turtle organizes the data sharing overlay on top of pre-existing user trust relationships. This protects the privacy of both data senders and receivers, as well as the intermediate relay nodes that facilitate the data exchange. Furthermore, Turtle is resistant to most of the denial of service attacks that plague existing peer-to-peer data sharing networks.

We have implemented Turtle and tested its performance. The implementation is efficient – both protocol overhead and CPU load are low. Thanks to the giFT framework we have a user-friendly GUI, multisource downloading, hash-links, file metadata and much more.

7.1 Future work

The Turtle, as presented in this thesis, was designed as a proof of concept network and does not include many features known from contemporary peer-to-peer networks.

7.1.1 Implementation

In order to have a real-world usable peer-to-peer application, it will be necessary to extend Turtle in many directions. The following list gives an overview of domains, where Turtle's lacks are most significant. Especially the first two of them complicate starting real Turtle network.

Firewalls It is a common problem of peer-to-peer networks, that most participants are protected by firewalls that block incoming TCP connections and limit outgoing connections to a small set of protocols (e.g.

HTTP). Statistics at [23] show that only about 20% of the total number of Gnutella nodes are accepting connections. This is not due to firewalls only, but they are certainly a contributor to this low number of hosts that accept connections. In order to support firewalled nodes we would have to modify communication protocol of Turtle. Modifications would affect only low-level TCP communication layer.

Dynamic IP addresses Recently it became common practise of ISPs (Internet Service Providers) to assign IP addresses to hosts dynamically via DHCP (Dynamic Host Configuration Protocol [13]). As a result, nodes change their addresses frequently, which does not comply with current Turtle implementation, where IP addresses of neighbours are configured statically. A partial solution to this problem is simple – a node, whose address has changed, can still connect to its neighbours and notify them about the change. Neighbours then update their configuration and start using the new address. This solution does not work if *both* neighbour nodes change their address at the same time.

Bandwidth management Our Turtle implementation employs a very simple bandwidth management scheme. It is possible to limit total bandwidth, which is evenly distributed among all Turtle TCP connections. However current bandwidth management does not support prioritizing of particular TCP connections, nor does support limits on separate virtual circuits. Because of that, we are not able to limit relayed traffic to some fraction of total traffic. We are also not able to limit query traffic in favour of file transfers. Another feature not implemented yet is calculating expected throughput of file transfers. This information should be present in query hit packets to allow users to make better decisions about which download source to choose, if they receive same query hit from different hosts.

Turtle has been released under GNU Public License at SourceForge web site as project `turtle-p2p`[31]. Source code is available via public CVS and all developers are welcome to contribute.

7.1.2 Protocol design

There are three main areas related to protocol design that could be subject to further research.

Security properties Although security properties of Turtle offer protection against many kinds of attacks, there is still space for improvements.

First vulnerability is caused by the fact, that file data, as they travel through Turtle network, reside temporarily on intermediary nodes in un-encrypted form. Even though it is only in memory (in communication buffers, see Figure 4.9), the memory might get swapped out to hard-disk, which could compromise the node. A simple solution of this problem is to encrypt file data at the source node and leave them encrypted throughout the transfer (in addition to encryption that is already performed at each hop). If a public/private key protocol is used, intermediary nodes have no way to determine what they are relaying, until they begin to actively intercept the protocol.

Second modification we are considering aims at reducing damage that can be done after security break in one correct Turtle node. Right now it affects the node itself plus node's neighbours. By splitting file data into shares (as in [32]) and transferring them through two different paths, we could possibly reduce the damage just to invaded node.

Query protocol It has been well documented elsewhere[29] that peer-to-peer networks cannot scale if they employ simple broadcast querying scheme. This is why ultrapeers were introduced in Gnutella network and FastTrack has search and index nodes. We will have to consider carefully how to extend Turtle's querying mechanism without harming anonymity of its users.

Economic model Because of the trust properties of Turtle's communication overlay (only nodes that trust each other directly interact), it is possible to enhance Turtle with an economic model that would encourage cooperation and sharing. For example, when sending back a query hit packet, a node can also include a price tag for supplying the given item, with each node in the broadcast tree adding its *relay fee* to the price tags received from its children. The query initiator can then use the final price tag as an additional selection criteria when deciding which data item to request. These payments can be aggregated over longer intervals, by having Turtle nodes keeping track of the amount owed to/owed by friend nodes. A node can then periodically report the "balance sheet" to its owner, who can settle the matter with his friends by out of band means (e.g. cash exchange).

Simulations In order to verify our expectations about functioning of the Turtle network, we will have to implement simulator of the network and run it on the Orkut graph data. In this way we can collect statistics about network traffic and measure data throughput and latency. We

can also tune flow control settings of Turtle or possibly reveal some design mistakes we have made.

Appendix A

Structure of packets

In this appendix, structure of packets used in the Turtle network is described. All data structures are stored in network byte order (big endian).

A.1 Virtual circuit command packets

All virtual circuit command packets have a fixed length (14 bytes) header part. Some commands do not require any other information, others have additional information attached to the packet header. Underlying secure TCP connection is used as a data stream, packets are not aligned to data blocks described in Section 4.2.2. If the protocol described here is violated in any way, underlying TCP connection must be closed immediately.

The CIRCUIT CONTROL packet gives the receiving side credits for opening more virtual circuits. This packet is usually sent immediately after the TCP connection has been established and then after a virtual circuit has been closed.

Length	Description
2	Command (1)
4	Circuit credits
8	Zero padding

Table A.1: The CIRCUIT CONTROL packet.

The CONNECT packet is a request to open new virtual circuit. It can be sent only if the sender has at least one circuit credit. The circuit segment ID must be unique for this TCP connection. Uniqueness is guaranteed by a simple rule for generating circuit segment IDs, which says that the node with

the higher node ID generates odd IDs and the node with the lower node ID generates even IDs.

Length	Description
2	Command (2)
4	Circuit segment ID
4	Destination address length
4	Source address length
2–8000	Destination address
2–8000	Source address

Table A.2: The CONNECT packet.

The CONNECTED packet can be sent only after CONNECT packet has been received.

Length	Description
2	Command (3)
4	Circuit segment ID
8	Zero padding

Table A.3: The CONNECTED packet.

The FORWARD packet transfers data through virtual circuit. Node is not allowed to send more data through the circuit than how many credits it has for this circuit.

Length	Description
2	Command (4)
4	Circuit segment ID
4	Data length
4	Zero padding
0–16384	Data

Table A.4: The FORWARD packet.

The FLOW CONTROL packet gives the receiving side credits for sending more data through specified virtual circuit. This packet is usually sent immediately after the virtual circuit has been established and then every time the node has processed (e.g. forwarded) data received from the circuit.

Length	Description
2	Command (5)
4	Circuit segment ID
4	Data credits
4	Zero padding

Table A.5: The FLOW CONTROL packet.

The CLOSE packet is a request to close the virtual circuit. No other packet except CLOSED packet can be sent after this packet.

Length	Description
2	Command (6)
4	Circuit segment ID
8	Zero padding

Table A.6: The CLOSE packet.

The CLOSED packet is a final packet of the virtual circuit. No other packet can be sent after this packet.

Length	Description
2	Command (7)
4	Circuit segment ID
8	Zero padding

Table A.7: The CLOSED packet.

The SAR ADDRESS packet informs the receiving side about new address of Service Address Resolver. This packet is usually sent immediately after the TCP connection has been established. Stateless anonymous address should be used.

Length	Description
2	Command (8)
4	SAR address length
8	Zero padding
4–8000	SAR address

Table A.8: The SAR ADDRESS packet.

A.2 Service Address Resolver packets

Packets described in this section are sent via virtual circuit established from arbitrary service to SAR. More requests can be sent on one circuit.

The service list request packet is sent to SAR to get list of services. SAR always replies with the service list reply packet.

Length	Description
4	Command (1)
4	Length (0)

Table A.9: The SAR service list request packet.

The service list reply packet is sent by SAR as a reply to the service list request packet. Service list might be empty.

Length	Description
4	Command (2)
4	Service list length
0–102392	Service list (concatenation of ASCIIZ strings)

Table A.10: The SAR service list reply packet.

The service address request packet is sent to SAR to get anonymous address of a service. Name of service is without trailing zero. SAR replies with the service address reply packet or with the unknown service packet.

Length	Description
4	Command (3)
4	Service name length
1–256	Service name

Table A.11: The SAR service address request packet.

The service address reply packet is sent by SAR as a reply to the service address request packet. Stateless anonymous address should be used.

Length	Description
4	Command (4)
4	Service address length
4–8000	Service address

Table A.12: The SAR service address reply packet.

The unknown service packet is sent by SAR as a reply to the service address request packet when requested service is not registered at SAR.

Length	Description
4	Command (0)
4	Length (0)

Table A.13: The SAR unknown service packet.

A.3 Query Service packets

Query Service packets are sent via virtual circuits established between Query Sender and Query Receiver of neighbour nodes. The circuits are permanently opened and used for transferring packets until one of the neighbours is shut down.

Query packet is sent from Query Sender to Query Receiver and contains query or possibly multiple versions of the query. A node evaluating the query should find the first version that has known type (i.e. the node understands how to evaluate the query). Currently there is only one query type (*basic*), but in the future we might add for example XML querying. In such case query packets would contain XML query as first, preferred, version and then simplified basic query. Nodes that would support XML querying could evaluate XML query, other nodes would evaluate basic query. Each query version begins with one byte query type followed by type-dependent data.

	Length	Description
	8	Query ID
	1	Packet time-to-live
	1	Query version count
	4	Total length of all query versions
for each version	4	Starting position of query version in data part
for each version	1–16384	Data of query version

Table A.14: The query packet.

Basic queries are transferred in the form of a parse tree. Query parse tree has *internal nodes* and *leaf nodes*. An internal node represents an expression with logical operator and has one or two subtrees. Expressions with binary operator (`AND`, `OR`) have two subtrees, expressions with unary operator (`NOT`) have only one (left) subtree. Right subtree must have zero length in that case.

Length	Description
2	Logical operator (see Table A.17)
2	Left subtree length in bytes
2	Right subtree length in bytes
0–	Left subtree
0–	Right subtree

Table A.15: Internal node of query parse tree.

Leaf node represents basic formula with attribute name, relational operator and possibly attribute value. Attribute value is missing for unary operators (`EXISTS`) and must have zero length. Note that simple query with no logical operator contains only one – leaf – node and no internal nodes.

Length	Description
2	Relational operator (see Table A.17)
2	Attribute name length in bytes
2	Attribute value length in bytes
0–	Attribute name
0–	Attribute value

Table A.16: Leaf node of query parse tree.

Table A.17 gives an overview of all operators that might be used in basic queries. Operator ID allows to distinguish between internal node and leaf node when reading basic query from a query packet.

Operator	Type	Arity	ID
AND	logical	binary	0x0001
OR	logical	binary	0x0002
NOT	logical	unary	0x4003
<	relational	binary	0x8004
<=	relational	binary	0x8005
==	relational	binary	0x8006
>=	relational	binary	0x8007
>	relational	binary	0x8008
=	relational	binary	0x8009
EXISTS	relational	unary	0xc00a

Table A.17: The query operators.

Query hit packet is sent from Query Receiver to Query Sender and contains information about single query hit. The information consists of set of attributes and their values followed by the anonymous address of File Sender, which can be used to download the file. Currently there are two mandatory attributes defined: “name” is name of the file and “length” is length of the file.

	Length	Description
	8	Query ID
	1	Packet time-to-live
	4	Expected download bandwidth in kbps
	2	Length of attribute section
	2	Attribute count
	4	Length of address
for each attribute	2	Starting position of attribute name in attribute section
	2	Starting position of attribute value in attribute section
		(attribute section)
for each attribute	0–255	Name of attribute
	0–255	Value of attribute
	4–8000	Address of file sender

Table A.18: The query hit packet.

A.4 File Service packets

File Service packets are sent via virtual circuit established between File Receiver and File Sender. Multiple requests can be sent on one circuit.

The file request packet is sent by File Receiver to get file data. Name of the file is without trailing zero. File Sender replies with the file reply packet.

Length	Description
4	Command (1)
4	Length of the whole packet
4	Position where to start download
4	Maximum bytes to download
1–255	Name of file

Table A.19: The file request packet.

The file reply packet contains the requested file data. Their length is never higher than what was requested – even if the file does not fit into reply packet. If any error occurs or if starting position is higher than the file length, the file reply packet must not contain any data in file data section.

Length	Description
4	Command (2)
4	Length of the whole packet
4	Error code
4	Position where the download starts
0–	File data

Table A.20: The file reply packet.

Bibliography

[1] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[2] Anonymizer. Official web site, 2004. http://www.anonymizer.com/.

[3] Apollon. The graphical client for the giFT framework. official web site, 2004. http://apollon.sourceforge.net/.

[4] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks, 2002.

[5] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981.

[6] Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[7] L. Cottrell. Frequently asked questions about Mixmaster remailers, 1996. http://www.obscura.com/~loki/remailer/mixmaster-faq.html.

[8] DAS2. The Distributed ASCI Supercomputer 2. official web site, 2004. http://www.cs.vu.nl/das2/.

[9] Gnutella Developer Forum. Official web site, 2004. http://groups.yahoo.com/group/the_gdf/.

[10] J. Douceur. The sybil attack, 2002.

[11] EDonkey2000. Official web site, 2004. http://www.edonkey2000.com/.

[12] FastTrack. Documentation of the known parts of the FastTrack protocol, 2004. http://cvs.berlios.de/cgi-bin/viewcvs.cgi/gift-fasttrack/giFT-FastTrack/PROTOCOL?rev=HEAD&content-type=text/vnd.viewcvs-markup.

[13] Internet Engineering Task Force. Dynamic Host Configuration Protocol (RFC 1541), 1993.

[14] Gnutella Developer Forum. The Gnutella protocol specification v0.4, 2004. `http://rfc-gnutella.sourceforge.net/developer/stable/index.html`.

[15] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, D.C., November 2002.

[16] Freenet. A distributed anonymous information storage and retrieval system. official project web site, 2004. `http://freenet.sourceforge.net/`.

[17] Friendster. Online community web site, 2004. `http://www.friendster.com/`.

[18] giFT. Project official web site, 2004. `http://gift.sourceforge.net/`.

[19] giFToxic. The graphical client for the giFT framework. official web site, 2004. `http://giftoxic.sourceforge.net/`.

[20] Gnutella. Community web site, 2004. `http://www.gnutella.com/`.

[21] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Hiding routing information. In *Information Hiding*, pages 137–150, 1996.

[22] KaZaA. Official web site, 2004. `http://www.kazaa.com/`.

[23] Limeware. Official web site, 2004. `http://www.limewire.com/`.

[24] Napster. Official web site, 2004. `http://www.napster.com/`.

[25] Orkut. Online community web site, 2004. `http://www.orkut.com/`.

[26] Bogdan C. Popescu, Bruno Crispo, and Andrew S. Tanenbaum. Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System. In *Proc. 12th Cambridge International Workshop on Security Protocols*. Springer-Verlag, April 2004.

[27] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[28] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, page 99. IEEE Computer Society, 2001.

[29] Jordan Ritter. Why gnutella can't scale. no, really, 2001. `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[30] Tor. An anonymizing overlay network for TCP. official web size, 2004. `http://freehaven.net/tor/`.

[31] Turtle. A peer-to-peer architecture for safe sharing of sensitive data. official project web site, 2004. `http://turtle-p2p.sourceforge.net/`.

[32] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[33] Rolan Yang. Georkut density map web site, 2004. `http://www.datawhorehouse.com/orkut/`.