Bachelor thesis
Computer Science

Radboud University

Efficiency of large-scale DC-networks

Author:
Moritz Neikes
4099095
m.neikes@student.ru.nl

Supervisor:
Anna Krasnova
anna@mechanical-mind.org

Supervisor:
Peter Schwabe
peter@cryptojedi.org

July 1, 2014

Abstract

DC-networks offer desirable properties for anonymous communication, but a practical implementation for large networks is challenging. Other approaches like *Tor* exist and are easier to use in large networks, but they do not provide strong anonymity.

This thesis describes a design for a DC-network implementation that targets large networks. It does not offer new methods to counter attacks, but it does overcome practical limitations and bottlenecks of classical DC-networks. A prototype implementation is provided which uses this design and which can be used in future research on large-scale DC-networks.

Scheduling has a crucial influence on the throughput of large DC-networks, since all clients share a single communication channel. A novel scheduling algorithm, called *Fingerprint scheduling*, is presented, which efficiently organizes access to this shared channel while being stable in the face of unreliable client connectivity. Using simulations, the performance of this algorithm is compared to the performance of existing scheduling protocols for up to $50,000$ clients. The results show that Fingerprint scheduling is the scheduling algorithm of choice for large DC-networks.

Contents

1 Introduction **5**

2 From the dining table to computer networks **9**
2.1 Key handling 13

3 Scheduling **19**
3.1 Fingerprint scheduling 19
3.2 Optimizing Fingerprint scheduling 23
3.3 Other scheduling protocols 28
3.4 Performance Comparison 31
3.4.1 Client activity 33

4 Implementation details **35**
4.1 Message handling 35
4.2 Throughput benchmarks 36

5 Possible improvements **37**

6 Related work **39**

7 Conclusion **41**

8 Acknowledgements **43**

A Details on measurement results **47**
A.1 Configuring Pfitzmann's algorithm 47
A.2 Exact measurement results 49

B Command line interface reference **53**
B.1 Build and run 53
B.2 Basic program flow control 53
B.3 DC-network commands 54
B.4 Debugging commands 55
B.5 Scheduling benchmarks 55

Chapter 1

Introduction

In times where processing power and network bandwidth allow for huge surveillance systems, it gets more important to be able to exchange information anonymously. There are widely used and reliable methods to encrypt a message, so that it becomes computationally difficult for an attacker to decipher its *content*. But in some cases, the identity of the communicating parties is what needs to be protected, not the content of their communication.

Anonymous communication can be desired in various situations. Whistle-blowers might want to protect their identity when communicating with journalists, or they want to anonymously contribute to platforms like WikiLeaks[1]. Also, digital currencies like Bitcoin do not offer anonymous payment right away[2]. In order to make the origin of transactions untraceable, payment informations have to be transmitted anonymously.

These applications require a communication technique that provides strong anonymity, even against attackers that are potentially as strong as a government. Dining Cryptographers' networks (DC-nets), as developed by David Chaum in 1988 [2], provide the desired properties. Members of a DC-network can transmit arbitrary messages, and as long as certain requirements are met, the identity of the sender of such a message is unconditionally secure. There are other approaches to achieve anonymity, which are based on Mix-nets. Tor [4] is a well-known implementation of such a Mix-net network. However, these approaches are in general vulnerable to traffic-analysis attacks [11]. Chapter 6 will discuss other approaches and existing DC-net implementation in more detail.

For a long time, DC-net implementations targeted small networks with no more than 40 participants [5, 3]. Although large networks offer desirable advantages, as discussed later in this chapter, little attempts have been made to implement DC-networks which support a much larger number of users. A more recent version of an implementation called Dissent, however, is able to handle networks with up to $5,000$ clients [15].

A practical implementation of large DC-nets is challenging for multiple reasons. Na-

[1] `www.wikileaks.org`, a platform that is used to publish (typically confidential) documents and e-mails.

[2] `https://bitcoin.org/en/you-need-to-know`; The official Bitcoin website, stating that Bitcoin payments are not anonymous.

tive DC-nets are vulnerable to Denial of Service attacks as well as certain attacks on anonymity. But even if all participants of a network are honest, some challenges remain: DC-nets naturally produce a huge amount of traffic, which grows linearly with the number of clients. The implementation has to distribute this traffic in such a way that communication is practical, given the bandwidth limitations of an average Internet connection. Another problem is the availability of clients. For the protocol to succeed, all clients have to be constantly connected to the network. In case that a client drops out unexpectedly, the outcome of the current protocol run will be corrupted. These drop-outs become more frequent, the more users are simultaneously connected to the network. The implementation has to handle this in a fast manner. Also, since all clients share a single communication channel, efficient scheduling is required in order to organize transmissions.

This thesis describes an implementation design that can be applied in a large-scale DC-network. The design consists of a client-server hierarchy that was proposed in [7], so that the network load can easily be distributed amongst several servers. It is also able to handle unstable client connectivity, since it can recover quickly from clients that drop out unexpectedly. Furthermore, a novel scheduling algorithm is introduced, which allows effective scheduling in large networks, where constant client connectivity can not be guaranteed. The scheduling algorithm will be compared to existing scheduling protocols. The prototype implementation can be obtained at `https://github.com/25A0/DCnet/archive/v0.3.zip`.

As a foundation, the next chapter will explain the basic principle of DC-networks along with commonly used terminology.

The Dining Cryptographers' protocol

To demonstrate the principle of DC-networks, this section explains how three cryptographers, Alice, Bob and Charlie, can use a DC-network to exchange a single bit of information[3]: Alice, Bob and Charlie are having dinner together (hence the name) at their favourite restaurant. At the end of the evening, when they want to pay and leave, they are told that someone already paid for their dinner. There are two possibilities: It either was one of them who generously paid for the dinner, or it was the NSA that paid. But in case it was one of them, they all want to respect the possibility that the payer does not want to give away their identity. Therefore, they come up with a simple protocol to exchange this one bit of information. By tossing coins, they create pairwise shared, secret keys for themselves and the neighbour to their right. They also individually choose a secret single bit that represents the message they want to send. This message is 0 for those who did not pay for the dinner, and 1 for the person that paid for the dinner. Let Alice be the one who paid for the dinner. In the next step, each of them secretly calculates the XOR of both shared keys and their message bit, and announce the result of this calculation. Finally, the result of the protocol is calculated by XORing the announced outputs. In this case the result is 1, since Alice inverted the combination

[3]This little story is essentially taken from [2, p. 1]

of her shared keys. If she had not done so, then each key would have been involved exactly two times throughout the whole protocol, which would have caused all keys to equal out, so that the result of the protocol would have been 0. Chaum proves in [2] that it is impossible for Bob and Charlie to identify Alice as the origin of the message.

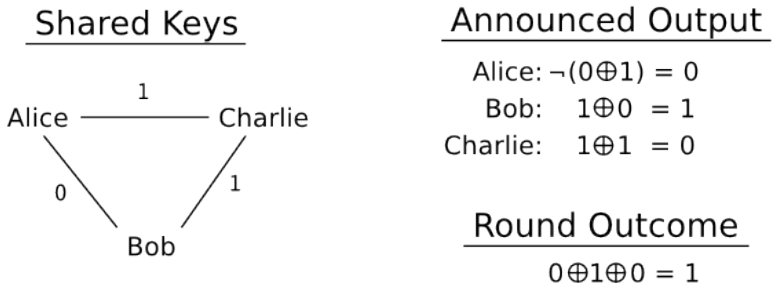

Figure 1.1: A toy example of the DC-network protocol.

Throughout this thesis, the following terminology will be used in relation to DC-networks: A *message* m is one bit which a user wants to transmit with the help of DC-net. The *output* O_i of user i is the result of XORing the message with all shared keys of that user: $O_i = m \oplus \kappa_1 \oplus \kappa_2 \oplus \cdots \oplus \kappa_k$. A *round* consists of three steps: All users publish their outputs O_i, all the outputs are XORed, and the result M is broadcast. The overall *outcome* M of a round is the result of XORing the output of all users, thus $M = O_1 \oplus O_2 \oplus \ldots \oplus O_c$, where c is the number of users. The outcome of a round is either a message or a collision of messages. If a user is choosing m to be equal to zero, this user is just *participating* in the current round of DC-net. Otherwise the user is *sending* in this round.

The protocol can easily be extended to support messages of multiple bits. To communicate a message of n bits, users share keys of n bits with each other. The output is calculated by XORing all shared keys bit by bit. If a user wants to send their message in a given round, the output is also bitwise XORed with that message. The outcome of the round is likewise the result of bitwise XORing the output of all users.

A native DC-net is vulnerable to several Denial of Service attacks and attacks on anonymity. There are existing implementations that provide solutions to counter these attacks. Herbivore and Dissent are two of them and will further be discussed in Chapter 6. This work, however, will not focus on resistance against attacks, but on efficiency.

Nevertheless, some attacks are related to the size of the network. An implementation that supports large networks can protect its users from these kind of attacks. In a large DC-network, a user can hide in a larger *anonymity set*[4], which protects them against statistical analysis and intersection attacks [5, p. 12]. Statistical analysis threatens the participants of small anonymity sets in the following way: Imagine an attacker that

[4]Following the definition in [8], *Anonymity Set* describes here the set of users in which a specific user can not be distinguished from other users by an attacker.

tracks at which times user u is part of the network, and at which times a certain web service is accessed. If the attacker discovers a strong correlation between those two events, then it is likely that user u accesses this web service. Large anonymity sets can help to hide these correlations. Intersection attacks target long-running transmissions. By tracking network membership during a long-running transmission, an attacker can narrow down the origin of this transmission to the subset of users that has continuously been connected to the network throughout the entire transmission. A large anonymity set helps to make this attack less effective.

This thesis consists of two major parts: Chapter 2 describes the design of the prototype implementation, and Chapter 3 introduces a novel scheduling algorithm for large DC-networks. Section 3.4 compares this algorithm to existing scheduling methods. Chapter 4 gives some details on the prototype implementation, as well as the results of minor benchmark tests. Chapter 5 presents two possible improvements that can help to increase the efficiency of the implementation. Chapter 6 discusses related work, and Chapter 7 summarizes the accomplishments of this thesis.

Chapter 2

From the dining table to computer networks

In [2], Chaum introduces the theoretical concept of DC-nets, as described in the previous chapter. He discusses some important issues of a practical implantation of DC-net. He proposes to slow down the transmission rate in times where no user wants to transmit data in order to not waste much transmission bandwidth during those idle periods. He also briefly explains how pseudorandom sequence generators can be used to produce a lot of key material based on a short shared secret key. However, many more details are required for an actual implementation. This chapter will describe how the provided prototype implementation is designed and discuss some important design decisions.

Chaum's description of the basic algorithm is brilliantly simple, but some changes and extensions are necessary as soon as coins get replaced by bits, and the dinner table gets replaced by the World Wide Web.

Packets. The provided prototype implementation is designed for stations that are connected via the Internet. A constant data stream would be the ideal environment for a DC-net communication medium, but Internet traffic is handled in packets. It would be a waste in efficiency to transmit one bit per message. Instead, each round transmits a fixed number of bits. Transmitting rounds with multiple bits at once is suggested by Chaum, too, as a requirement to use existing communication techniques for shared channels [2, p. 5]. His ideas will further be discussed in Section 3.4.

The size of a packet should be chosen based on the target application of the whole network since the average length of a message heavily depends on the application: If used for twitter or text messaging, even 256 byte packets might be enough to encapsulate a single message, but already a chat message could easily exceed this limit. If the DC-net is used for file exchange, even packets of multiple megabytes might still be reasonable. Small packets will increase the relative overhead that is added by the 40-byte header of the TCP/IP protocol, while with large packets there is a high risk that space is left unused.

In general, the efficiency of the network will rise, the better the packet size fits the

average length of a message. In the prototype implementation, packets have a default size of 1024 bytes. This size was chosen as a compromise between reasonable message size and communication overhead.

In case that a message is too big for a single packet, users could add identifiers to link different parts of their packets together. Note that this feature is not implemented in the prototype implementation.

Packets do not only contain the message payload itself, they also hold additional data that is used to organize the communication of the network. Figure 2.1 illustrates the different sections of a packet. The header (red) contains the number of the round that this packet belongs to. This number cycles through a fixed number range. The payload (blue) contains scheduling data as well as the message itself. The entire payload is encrypted with the shared keys, while the header is transmitted unencrypted.

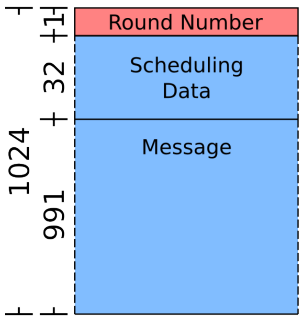

Figure 2.1: A schematic overview of the different sections of a packet. The size, in Bytes, of each section, as well as the overall size of the packet, are denoted on the left-hand side.

A message might not always fill the entire message section. Therefore, 10-Padding is applied by placing a `1` right after the end of the message and filling the remaining space of the message section with `0`s. Figure 2.2 illustrates this procedure. When reading the message in reverse, the first `1` will separate the actual message payload from the padding payload. This allows to determine the exact size of the message reliably, while adding minimal overhead. Note that not the entire packet capacity is used for message payload – at least the very last byte is always reserved for padding, even if the payload would otherwise fill the entire packet.

With these changes, the individual packets will make efficient use of the available network bandwidth. The next step is to define how stations exchange those packets in a network like the Internet.

Client-server structure. Chaum briefly explains how DC-net could be applied in a computer network [2, p. 5]. Back in 1988, it was common to have ring networks, where each participant is directly connected to their adjacent neighbours. In such a ring network, packets are forwarded to adjacent participants until they reach their destination.

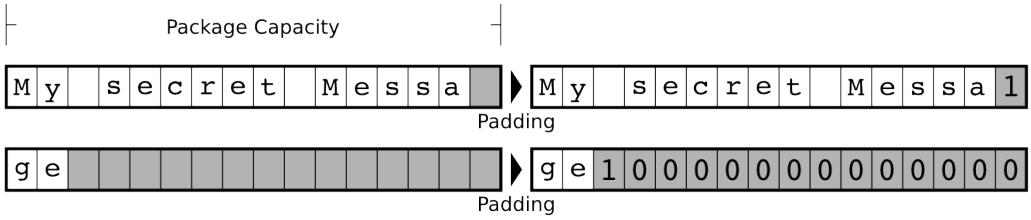

Figure 2.2: 10-Padding is applied to the message payload of two different rounds.

Traceable traffic travels on average through half of the nodes until it reaches its destination. Messages in a DC-net, however, would travel through each node twice before the round is completed:

> "Each participant exclusive-or's the bit he receives with his own output before forwarding it to the next participant. When the bit has traveled full circle, it is the exclusive-or sum of all the participants' outputs, which is the desired result of the protocol. To provide these messages to all participants, each bit is sent around a second time by the participant at the end of the loop." [2, p. 5]

Nowadays, there are hardly any ring networks in use. Local area networks usually have a star-topology with one shared gateway to the Internet. The topology of the Internet itself is hard to describe with simple topology models[1].

Instead of simulating a ring network with token-ring protocols or similar methods, our implementation bases the network communication on a hierarchical client-server system: Clients transmit their output to a server, this server then combines the incoming outputs and sends the outcome back to the clients to complete the round. This has some clear benefits over simulated ring networks: In a real ring network, it is obvious which nodes are adjacent to each other, while in a simulated ring network, this has to be decided software-wise. The same goes for the order in which stations transmit their output. In a client-server network, the server forms a central melting pot for the output of all stations, where it is not necessary to determine an order in which stations send their output.

Of course, with this network topology, the server needs a much faster network connection than the clients that are connected to it[2]. If there are more clients than a single server can handle, the client-server network can be extended to a multi-layer, tree-shaped graph, where clients form leaves in this tree and servers form nodes. Clients connect to one of multiple servers. The servers all combine the received messages, and then forward them to the next node that is closer to the root. This is repeated until the

[1] A research by Siganos et al. describes the Internet topology with the so-called *Jellyfish* model [12]

[2] In general, if the lowest connection bandwidth of all connected clients is b Bytes per second, and there are c clients connected to a server, then the server requires a connection bandwidth of $c \cdot b$ Bytes per second, otherwise the server forms the bottleneck of the throughput.

messages reach the single server at the root of the tree. This server then calculates the outcome of the round. This outcome is sent back to all stations in the network, layer by layer. This mechanism allows to install multiple servers that can flexibly distribute the network load amongst each other. This improves the bandwidth utilization since the network throughput is limited by the slowest client, rather than by the connection speed of a single server.

It should be noted that Pfitzmann has presented this approach in [7].

Another crucial point is the availability of clients: Stations may drop out unexpectedly, especially when connected via mobile phones [6]. In a client-server network, the server can keep track of the connectivity of each client, and can inform the remaining network members in case a client drops out. In a ring network, each client would have to draw their own conclusions in case that one of their neighbours does not react in time. It might be that particular neighbour who lost connection to the network, or the delay is caused by a node further down the ring that is not responding. This makes a ring-network hard to maintain when there are a lot of clients. Meanwhile, a central server can reliably detect which of their clients is causing delay.

Just as the ring network, a client-server network with n clients requires $2n$ messages to be transmitted per round: Each client has to send their output to the server, and the server replies to each client with the outcome of the round. A general proof in [5, p. 7] shows that a DC-net with n nodes needs to transmit at least $2n-1$ messages to propagate a single message anonymously. In our approach, the server does not actively participate in the messaging, and does thus not count as a node. Therefore, each anonymously sent message requires at least $2n$ messages to be sent. Since our implementation targets networks where the number of clients is much bigger than the number of servers, this difference is negligible.

Structures with more than one central server require more messages to be transmitted for each round. The network shown in Figure 2.3 requires 24 messages for 9 clients. This further increases the number of bits that have to be transmitted for each anonymously sent bit. On the other hand, networks like these allow us to easily spread the load of the whole network among multiple servers.

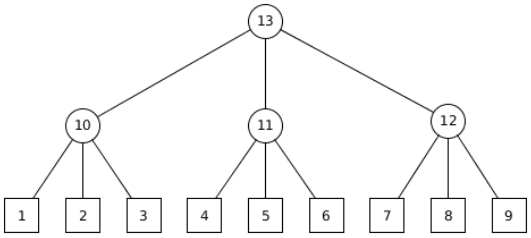

Figure 2.3: A hierarchical network with four servers and nine clients.

Network status. While sitting at a dinner table, all members are aware of each other – they can see and identify each other, and they can use the cover of their menu to

create shared secret channels. When stations are connected via the Internet, there is not such an intuitive way to find and identify other stations. Therefore, stations need to communicate changes in the network. In the prototype implementation, this is solved by having stations sending *Network Status* packets back and forth. The identity of stations is represented by aliases. Aliases are necessary for stations to recognize one another in a network. In the default configuration, aliases are limited to 8 bytes.

So, what happens when a station wants to join an existing network? Joining a network happens in two stages. When a station s connects to a server, it will not automatically be part of the network. However, the server will automatically send a list to s that contains the aliases of all stations that are currently part of the network. Station s will also receive the *outcome* of all future rounds. As soon as the station wants to join the network, it informs the server about this. The station also announces the alias that it wants to use in order to be recognized by others. The server will broadcast this change, and as of the following round, station s will be a member of the network. This two-stage mechanism allows stations to join a network seamlessly even if the network is large. If there are thousands of stations on the network, then it might take seconds until s has received the entire list of aliases. If this delay was to occur within the normal protocol, it would create an unnecessary delay. With the two-step mechanism, this delay is shifted to a point in time where it only affects the joining station, rather than the entire network.

If a station leaves the network, then this change is likewise spread to all remaining stations. But this event affects the current round rather than the following one. In case that a station s drops out involuntarily, the outcome of the round will be corrupted. To overcome this, all remaining stations retransmit the messages that they intended to send in the current round, but with fresh key material. Stations that share keys with station s will not include those keys in the second transmission.

Currently, the prototype implementation does not distinguish between a station that involuntary or purposely leaves the network. A reasonable change would be to have stations announce in which round they want to leave. The upcoming change could be broadcast in advance, so that there is no need to repeat a round. However, repeating a round can not be avoided in case that a station drops out involuntarily.

2.1 Key handling

The remainder of this chapter will explain how keys are handled in the prototype implementation.

Key graph

The anonymity of every message is protected by the keys that stations share with one another, and it is threatened by colluding stations. Colluding stations are the stations under control of an attacker A. If a station s shares keys with colluding stations only, then A can reveal the origin of all messages that s sends. Since keys have such a crucial

influence of the anonymity of a message, stations should decide whom they want to trust. In this case, trusting a station only means that this station is not suspected to collude with an attacker; the anonymity of their messages stays the same, whether or not two stations trust each other.

Each station owns a key ring which contains all keys that this station shares with other trusted stations. As soon as a station sees a trusted station on the network, it will use the respective shared key to generate its output. Note that each additional shared key requires additional computations for each round. Especially for mobile devices, it might be desirable to keep the number of shared keys low. Otherwise the computational power of the device might become the bottleneck of the throughput of the entire network.

Users have to share keys with at least two other honest participants on the network to keep their messages untraceable. Chaum has proven in [2] that two keys are sufficient to achieve untraceability.

The prototype implementation contains a protection mechanism that prevents stations from sending meaningful messages whenever less than two keys can be used to encrypt the output. This mechanism guarantees a minimal anonymity set for all participants.

One could suggest that users who share keys with less than two other participants simply send empty messages until a second trusted user joins the network. However, this alone could compromise the anonymity of the station that shares a key with this user. Imagine a DC-net that is set up as illustrated in Figure 2.4. Bob shares secret keys with Alice and Charlie, which might give him the impression that his messages will be anonymous. However, since Alice and Charlie each only share a key with Bob, their output o_a and o_c will simply be the key that they share with Bob. Eve can now compute $o_a \oplus o_c$ and is left with k_b, the combination of Bob's shared key bits for this round. If she now was to compare this result to Bob's actual output o_b, she will unveil the message that Bob sent in this round: If Bob's output differs from the k_b, then Bob transmitted `1` in this round.

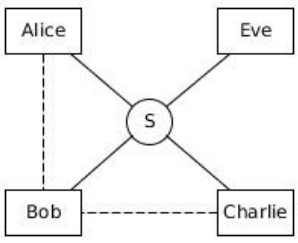

Figure 2.4: A DC-net in which Eve can potentially reveal Bob as the origin of messages. Dashed lines show the key graph: Alice and Bob, and Bob and Charlie share keys with each other.

The key graph must be assumed to be publicly known [2]. Therefore, Eve can assume that Alice and Charlie will only send empty messages, otherwise their anonymity against

Bob would not be guaranteed.

In the described scenario, Bob shares keys with two other participants, but his messages are still traceable. In order to avoid situations like these, each station could share a key with the server that it is connected to. It is important to note here that this requires a client to trust the server as much as the client trusts any other client that they share keys with. If we adapt the previous scenario in this way, then Alice will also share a key k_{as} with the server. Thus, o_a will then be $k_{ab} \oplus k_{as}$. Since k_{as} remains secret, Eve will not be able to identify Bob as the origin of a message. The same is accordingly true for Charlie. The key that each station shares with the server is treated just as any other key. This means that Alice and Charlie each share keys with two other stations now; they can send messages anonymously just as Bob does, and Eve could not identify the origin of their messages. The server could not do so, either, since Alice and Charlie each still share a secret key with Bob, which the server does not know. It should be noted that, even if a station shares a key with the server, it still needs to share at least one other key with a different station in order to send anonymously. Otherwise the server could still identify that station as the sender of the message.

There are other ways to solve this problem. Bob could instead analyse the key graph to find weak spots like Alice and Charlie. Bob would have to repeat this check every time that users connect to or disconnect from the network. Therefore this approach scales poorly with respect to the number of shared keys and clients.

This seems like a lot of effort for stations which, in the end, might not even be able to send messages before a trusted station joins. But it helps to avoid *deadlocks*: A group of stations in which all members only share keys with each other might not even be able to join the network at all. They would indefinitely keep waiting for each other to join.

Nonce initiation

The described key handling mechanism has currently one problem that needs to be addressed before it can be used in production: When a station recognizes a trusted station on the network, it will start to use the key that those two stations share. However, the nonce will initially be 0 – there is no mechanism implemented that determines an initial value for the nonce. This means that the shared keystream between two stations will repeat as soon as they get connected for the second time. This makes their messages potentially traceable.

Generating bit-strings

Keys in a DC-network must never be re-used, similar to the way one-time pad systems work. Each round requires new, unused key material. However, sharing random bit-strings becomes impractical for a long-lasting conversation. This section will explain how stations can use continuous bit-strings as keys that are generated from a single shared key.

Pseudo-random number generators (PRNG) can be used to generate continuous bit-streams locally. The generated bit-stream depends on the *seed* that is used to initialize

the PRNG. This allows two stations to generate the exact same bit-stream by using the same seed, namely the secret shared key. In this implementation, the stream cipher *Salsa20* [1] in PRNG mode is used. In each round, a new random bit-string is generated from the combination of the shared key and a nonce. Both stations increase the nonce by 1 after each round.

The amount of available key material is limited by the amount of available nonces. Once there are no unused nonces left, the stations will have to agree on a new shared secret key as the seed of the PRNG. The chosen implementation uses nonces with a size of up to 64 bit, allowing up to $2^{64} \approx 1.8 * 10^{19}$ possible nonces. It is worth demonstrating that in DC-net this amount of nonces is sufficient to generate bit-strings from the same seed for a long time, even if available bandwidth is unrealistically high. First, we will calculate how much key material each client needs per second. Based on that we can determine for how much time the available nonces will last. Let each packet be of 1024 bytes size. For a single round, the server and each station exchange two of these packets – the station sends its output to the server, and the server sends back the outcome of the round. Note that only the packet that the station sends to the server needs key material. Assume that all stations on the network are linked to the server by symmetric point-to-point connections with a bandwidth of 1 $TiByte/sec$ (one Tebibyte[3] per second). This bandwidth exceeds the upcoming Google Fibre[4] links approximately by a factor of 8000. Assuming that there are no interruptions or delays, the network would achieve a throughput of 512 rounds per second:

$$\frac{1,048,576\ Byte/sec}{2 * 1024 Byte} = 512 \frac{1}{sec}$$

As stated, only half of those packets are encrypted by the station, which means that a station needs 256 nonces per second. We can now estimate for which time period a station could use the DC-net constantly without running out of unused nonces:

$$2^{64} \div 256 \frac{1}{sec} \div 60 \frac{sec}{min} \div 60 \frac{min}{h} \div 24 \frac{h}{day} \div 365.25 \frac{d}{year} \approx 2,284,931,318\ years$$

Once two stations have agreed on a shared key, the number of possible nonces provide enough key material to communicate constantly for more than a billion years.

Sharing of random seeds for PRNG between stations is out of scope of this work. The usual way to solve this issue is to utilize public-key cryptography.

The described design introduces the changes that are necessary when DC-net is run in a network like the Internet. Thanks to the hierarchical client-server structure, network load can easily be distributed amongst all servers, which improves bandwidth utilisation. Furthermore, the design provides an efficient mechanism for clients to join the network,

[3] `http://en.wikipedia.org/wiki/Tebibyte` $1\ TiB = 1024 * 1024 * 1024 * 1024\ Bytes = 2^{40}\ Bytes$.

[4] `http://en.wikipedia.org/wiki/Google_Fibre` Google Fibre is a service that provides broadband Internet connection to consumer customers. It is currently (May 2014) possible to achieve a symmetric bandwidth of one gigabit per second.

and it remains stable despite unreliable connectivity of clients. Also, clients have full control over the keys that they share with other participants. But at the same time, they are prevented from sending meaningful data whenever the number of shared keys would threaten the anonymity of their messages. The next chapter will describe the last building block of an efficient large-scale DC-net: The scheduling algorithm.

Chapter 3

Scheduling

In a DC network, all users share the same communication channel. Whenever multiple users try to use that channel at the same time (i.e., by sending a message in the same round), a collision occurs. This leads to corrupted messages which have to be retransmitted. The goal of a scheduling protocol is to organize message transmission so that collisions are avoided. Scheduling has a crucial influence on the throughput of a DC network. In large networks, efficient usage of the DC-net protocol is even impossible without a proper scheduling algorithm, since constantly occurring collisions would cause a collapse of the network throughput.

The next section will present a novel scheduling protocol, called *Fingerprint scheduling*. This protocol provides efficient scheduling in the case of unreliable connectivity. It scales linear in the number of active clients and is therefore suited to be used in large networks with thousands of clients. Clients are called *active* whenever they want to transmit data. The following section explains how the protocol works and describes its properties. Towards the end of this chapter, other scheduling protocols will be discussed. Some of them are suited to be used in large networks, others are not. This chapter will conclude with a comparison between the suited protocols and Fingerprint scheduling.

3.1 Fingerprint scheduling

The first part of this section will explain the principle behind the novel scheduling protocol, while the last part will explain its properties.

A basic idea of many scheduling algorithms is to have users reserve rounds in advance, which they can then use to reserve their messages. Rounds that a user can reserve are called slots. A special reservation round is executed that allows users to voice their reservation attempts. In order to reserve slot x, the user transmits message 1 as the xth bit during the scheduling round. The outcome of the scheduling round shows 1 for each slot that was reserved, and 0 for all free slots [2, p. 5].

In this basic protocol, each slot is represented by a single bit. Unfortunately, collisions are impossible to detect whenever an odd number of users tries to reserve the same slot. Also, reservations will fail whenever an odd number of users tries to reserve the

same slot. In order to avoid this, one could use more slots. The DC-net implementation Herbivore uses this approach to avoid collisions during the scheduling process. Fingerprint scheduling is based on a different approach by using multiple bits to represent a slot. Before diving deeper into the functioning of Fingerprint scheduling, let us define some terms that will be used in the description of the algorithm:

A scheduling block is the part of the packet designated for scheduling.

A slot represents one round in the scheduling block.

A cycle is the interval of rounds that is planned in the schedule. If the schedule contains 8 slots, then the scheduling cycle lasts for 8 rounds. The clients have 8 rounds to find a consistent schedule.

A fingerprint is a random value that users use in an attempt to reserve a slot. The fingerprints are changed every round.

By using multiple bits to represent each slot, users can choose from multiple possible messages to mark their reservation attempt: For n bits per slot, the user can choose any message $f \in \{0,1\}^n \setminus \{0\}^n$ to mark their reservation[1]. This value f is called *fingerprint*. By having multiple bits per slot, conflicting reservation attempts result in more diverse collisions.

At its core, the slot reservation is performed in the following way: A user wants to reserve round n in the upcoming cycle. To voice this attempt, they set the value of slot n in the scheduling block to a freely chosen fingerprint f. If no other user tried to reserve the same slot, then the outcome of the round contains fingerprint f in slot n. Otherwise slot n will contain a collision of f and one or multiple foreign fingerprints. In this case the user can choose to either stick to the current slot, try to reserve a different slot, or withdraw their reservation attempt. Since the outcome of the round is broadcast, all users see which slots contain reservation attempts, but they can not unveil the identity of the users that reserve a slot. Other users will not attempt to reserve a slot that is already reserved. Nevertheless, once a user reserved a round, they have to repeat placing a value in the corresponding slot until the end of the current cycle. At the end of the cycle, the content of the scheduling block shows which rounds in the next cycle are reserved. A slot containing 0 represents an unreserved round. Every other value represents a successful reservation, or a collision of two reservation attempts. The following section will address the choice of the fingerprint that is used to mark a reservation, since it is crucial to the anonymity of a sender's identity.

Fingerprints

If not handled correctly, the fingerprint that a user chooses in order to mark their reservation attempt can leak information about their identity. If a user was to repetitively use

[1] This is every bit sequence of n bits except for a sequence of n zeroes.

the same fingerprint across multiple rounds, it would be possible to link their messages together, which can give away information about their identity in the long run.

There is yet another reason why it matters which fingerprint a user writes to a slot: Reservations can – just as message payload – collide as soon as more than one user tries to reserve the same slot in the same round. Imagine the following scenario: Three users use the same fingerprint x to mark their reservation. Also, all three users try to reserve the same slot n in the same round. This will lead to a collision that none of them can discover. When the output of that round is combined, two of the fingerprints will cancel out, but the third will persist. As a result, all three users will find their fingerprint x in slot n. They repeat writing x to slot n until the end of the cycle and each time their reservation attempts will collide in the same way. At the end of the cycle, they will all assume that their reservation succeeded. They will all use the round n to send a message, and their messages will collide.

There is a simple method to avoid those scenarios: Each round, a user chooses a new, randomly generated fingerprint. This significantly lowers the chance that a collision stays undiscovered during the reservation phase. Later in this chapter, we will see a more general description of the collision scenario, along with measurements about the likelihood of this scenario to occur.

Fingerprint scheduling protocol

This section presents the Fingerprint scheduling protocol. The steps of the protocol are written for a cycle of r rounds, starting in round 0.

The User:

1. Picks a random slot $\mathtt{s} \in \mathbb{N}, 0 \leq \mathtt{s} < \mathtt{r}$.
2. Generates a random fingerprint $\mathtt{f} \in \mathbb{N}, 1 \leq \mathtt{f} < 256$.[2]
3. Writes f to slot s, compute the output for this round as usual and broadcast it to the network.
4. For each round of the remaining scheduling cycle:
   (a) Waits for the result of the last round.
   (b) Compares the value of slot s with the current fingerprint f to check for a collision. If the slot does not contain the fingerprint, then there has been a collision.
   (c) Generates a new, random fingerprint $\mathtt{f}' \in \mathbb{N}, 1 \leq \mathtt{f}' \leq 255$ to replace the previous one.
   (d) If there was no collision, writes the new fingerprint f' to slot s to continue his reservation.

[2]The upper bound 256 comes from the fact that we use one byte (8 bits, 2^8 possible values) per slot in the schedule. Also, 0 is not a valid fingerprint since it is used to mark a free slot.

(e) If there was a collision, tosses a coin.

On *heads*, withdraws the reservation attempt for this slot. Instead, tries to reserve a different slot: Scans the scheduling block of the last round. If there is one or more free slot, picks one of them randomly, updates s to the index of that free slot, and places the current fingerprint f' in this slot for the next round. If there are no free slots left, withdraws any reservation attempts for the remainder of this cycle.

On *tails*, writes f' to s again, even though there was a collision, hoping that the other user will move away from this slot.

5. If the last round of the current cycle contains the current fingerprint f in slot s, the user assumes that the reservation of slot s succeeded. In the following cycle, the user uses round s to transmit their message. If there was a collision in the last round, the user withdraws the reservation attempt in order to avoid collisions.

In the prototype implementation, the scheduling is integrated into the general DC-net protocol, as the beginning of each packet is designated to scheduling information. The protocol could as well be executed in a separate scheduling phase, where packets consist only of scheduling information. However, Section 3.2 will demonstrate that Fingerprint scheduling mostly works best with 16 rounds per cycle, which leads to 16 bytes of scheduling data that has to be transmitted per round. If clients were to send packets that purely consist of these 16 bytes of scheduling information, then the 40 bytes header of the TCP/IP layer would add massive overhead. It is therefore strongly recommended to integrate Fingerprint scheduling in the general DC-net protocol.

It should be mentioned here that the number of users is not directly correlated to the number of slots. Unlike many other scheduling algorithms, even in a network with thousands of users, they will all try to reserve one out of 16 slots. The frequency with which their reservation attempts succeed is based on statistic distribution.

Properties of Fingerprint scheduling

The described protocol is suitable to be used in large DC networks. If configured correctly, Fingerprint scheduling remains stable even for thousands of users that try to transmit simultaneously. The constraints of an optimal configuration will be discussed in the next section. Fingerprint scheduling is in particular capable of dealing with unreliable connectivity of clients, without compromising efficiency. If a station drops out in the middle of the protocol, then any reservation attempt of this station will vanish by the next round, so that other stations can occupy the released slot.

The design of the Fingerprint scheduling protocol reflects the intention to use it in large networks. Clients are not allowed to reserve more than one slot per cycle. This choice was made to reduce the initial number of colliding reservation attempts that have to be resolved by the protocol. In small networks, however, some slots will remain unused. If there are 16 slots available for reservation, and only four users want to transmit data, then those four users will at most occupy a quarter of the available slots.

In general, if s slots are used per scheduling cycle, then the protocol will behave as if at least s clients are trying to transmit data simultaneously. In case that the protocol is used in a small network, the efficiency can be maximized with the following change: Given a network with c clients and s slots per scheduling cycle, all clients are allowed to each reserve $\lceil s/c \rceil$ slots. Clients will automatically fill up slots that are otherwise guaranteed to be unused. However, with this method it is still assumed that all clients are constantly willing to transmit data. Imagine a network with 32 slots per cycle and three clients, out of which two clients currently want to transmit data. The active clients will each try to reserve $\lceil 32/3 \rceil = 11$ slots, but the third client will not reserve any slots. Therefore, 10 out of 32 slots will remain unused. Depending on the implementation, this does not necessarily have to be a problem since the empty rounds could be skipped in the following cycle, but the active clients would still miss out on some slots that they could otherwise have scheduled right away. In its current state, Fingerprint scheduling is not the protocol of choice for small or inactive networks.

The frequency with which a reservation attempt succeeds is based on statistical chance. For example, in a network with 32 active clients and 16 slots per cycle, each client will on average succeed in every other reservation attempt. Although the slots are theoretically uniformly distributed, a client is not guaranteed to gain a slot at a fixed frequency. To make up for that, clients could change the withdraw chance according to the success quote of previous reservation attempts. If a client successfully reserved a round, then the following reservation attempt will have a higher withdraw chance. Similarly, if a client did not succeed in a single reservation attempt for a long time, it might help to raise the withdraw chance. This mechanism can help to uniformly distribute the slots among all clients. Note that dynamical changes of the withdraw chance are not part of the prototype implementation in any form.

There are different parameters that can be tweaked in order to change the exact behaviour of the algorithm. In the next section, we will compare the performance of the algorithm with different values for those parameters. At the end of this chapter, it will be shown that the amount of data that a client has to send in order to reserve a slot scales linearly with respect to the number of active clients.

3.2 Optimizing Fingerprint scheduling

There are two parameters of the Fingerprint scheduling algorithm that can be tweaked in order to optimize its scheduling performance. This section will first describe the meaning of those parameters and how they influence scheduling performance. After that, the performance of different configurations is compared, resulting in an optimal configuration for large networks. The last section of this chapter will then compare the performance of other scheduling algorithms to the performance of Fingerprint scheduling.

Fingerprint scheduling targets huge networks with potentially thousands of users that want to send simultaneously. A network of this size is hard to reproduce for testing purposes – instead, simulations will provide the benchmark measurements. Simulations allow us to test how the different algorithms react on up to $50,000$ stations sending

simultaneously.

The performance of Fingerprint scheduling and other scheduling algorithms changes significantly when moving from hundreds of clients to thousands of clients, or even beyond. To illustrate the changes properly, some plots in this chapter will (partly) use logarithmic axes. This affects the interpretation of growth. When a logarithmic scale is used for the x-axis while the y-axis features a linear scale, a linear growth of the measurements will appear as an exponential growth. Appendix A lists the exact measurement results, and describes in detail how to reproduce those measurements.

If not stated differently, each benchmark simulates a network where all clients constantly try to reserve a slot in order to send data. The influence of client activity is covered in Section 3.4.1.

Rounds per cycle (r): Fingerprint scheduling is executed in cycles with a fixed number of rounds. Let r be the number of rounds per cycle. The longer a cycle lasts, the more opportunities exist for clients to discover collisions in the slot that they are trying to reserve, in which case they could try to switch to a slot that is not occupied. As soon as there is at most one client left in a slot, this slot is called *collision-free*. The schedule has *converged* as soon as all slots are collision-free.

Fingerprint range (b): Fingerprints are picked from a predefined number range. Let b be the number of bits that are used to store a fingerprint in the scheduling block. A client can then choose from $2^b - 1$ possible fingerprints[3].

Comparing different configurations

With most scheduling protocols comes a certain amount of overhead, and Fingerprint scheduling is no exception here. In order to organize transmissions, all stations have to exchange additional data. Therefore, the scheduling performance is here measured by the amount of data that a client has to send in order to successfully reserve a slot. The term *data volume profile* will refer to this statistic. The optimal configuration aims to minimize this data volume profile. Therefore, the optimal configuration can be found by looking at the influence of r and b on the data volume profile.

Figure 3.1 shows the number of rounds that are necessary for the schedule to converge, for different values for b. A reading of 40 means that after 40 rounds the schedule has converged and all slots are collision-free. The chart shows that the choice of b has a significant influence on the speed in which the schedule converges. By increasing the number range by just one bit, the schedule converges much faster.

Convergence can be sped up significantly by increasing b. It is worth pointing out why it is important to let the schedule converge prior to the end of the cycle.

Fingerprint scheduling is based on a discussion mechanism that lasts several rounds. In large networks, multiple stations will try to reserve the same slot, which leads to collisions in the schedule. The stations detect those collisions and react accordingly.

[3]The fingerprint 0 is reserved to mark an empty slot

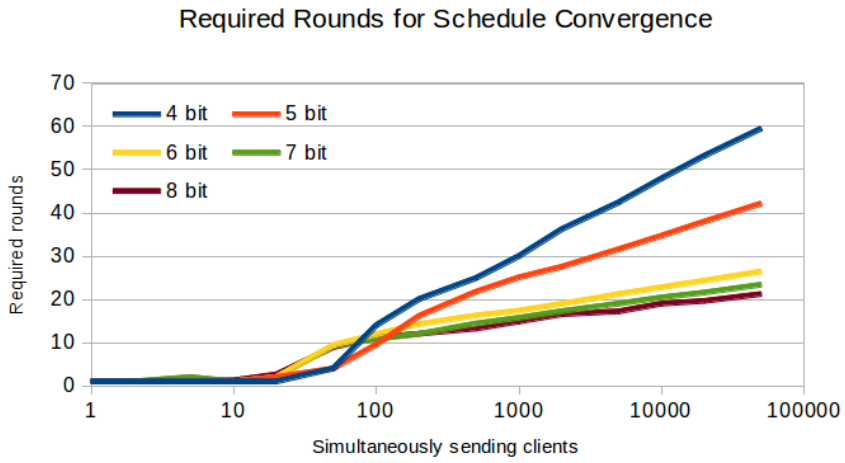

Figure 3.1: The number of rounds that are necessary for the schedule to converge, for different values of b. Table A.1 holds the exact measurement results of this chart.

Some of those stations will withdraw and move away from that slot, while one station will remain in the slot until the end of the cycle, and thereby will successfully reserve that slot. For this mechanism to work, it is crucial to give all stations enough occasions to resolve collisions.

The protocol defined in Section 3.1 causes a station to automatically withdraw its reservation attempt if a slot appears not to be collision-free by the end of the cycle in order to avoid message collisions. Some slots will therefore remain unused if the schedule does not converge in time. Figure 3.2 shows the number of unused slots by the end of a cycle of 32 rounds.

Depending on the way Fingerprint scheduling is implemented, empty slots do not necessarily have to be an issue. When the scheduling protocol is executed in a separate block, then empty slots could simply be skipped when the schedule is executed. But regardless of the implementation, if the slot remains unused eventually, then all the bits that have been transmitted in an attempt to reserve the respective scheduling slot will be wasted.

However, a high number of rounds per cycle is not the key to ensure convergence. Under certain circumstances, fingerprints can collide in a way that makes it impossible for clients to detect this collision. An undetectable collision occurs if $m \in \{x \geq 3 | (x+1) \div 2 \in \mathbb{N}\}$ participants[4] attempt to reserve the same slot, and all of them are using the same fingerprint. In this case, $m-1$ fingerprints will cancel out since $m-1$ is an even number. The remaining fingerprint will persist, causing all m participants to believe that their reservation succeeded.

There are also partially detectable collisions. To give one out of many possible examples, let there be exactly three clients that try to reserve a certain slot s. Two

[4]That is, all odd numbers starting from 3: $\{3, 5, 7, 9, \dots\}$

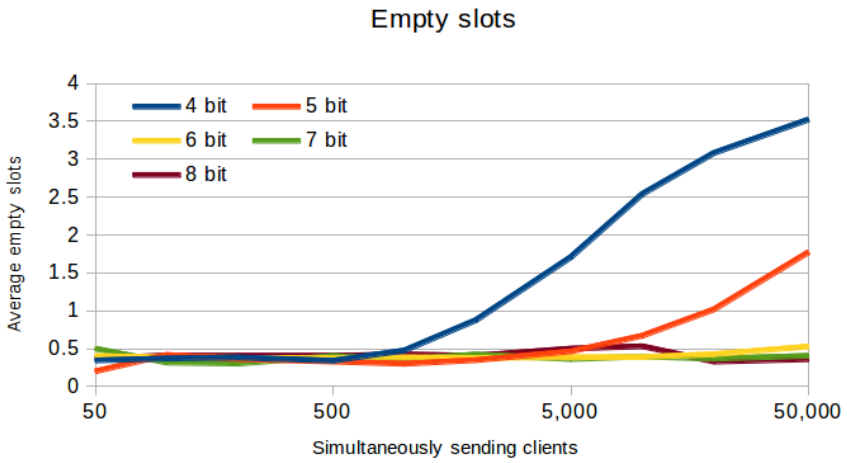

Figure 3.2: The number of empty slots after 32 rounds for various numbers of clients. See Table A.2 for exact measurement results.

of them use the same fingerprint f, while the third client uses fingerprint g. Since f appears two times, the bits will cancel each other out, so that the outcome of the round only shows g. The collision is detectable for the two clients with fingerprint f, but it is undetectable for the client with fingerprint g.

Both collision types cause a delay in scheduling convergence; clients have to be aware of collisions in order to react on them. The number range of fingerprints heavily influences the chance of undetectable and partially detectable collisions. Figure 3.3 shows the number of undetected collisions after 32 rounds for different numbers of bits. A collision counts as undetected if multiple clients reserve a slot and each of them believes that no other client is trying to reserve that slot. A reading of 10 means that, after 32 rounds, 10 different clients believe that they successfully reserved a slot, where, in fact, there is at least one other client that believes to have successfully reserved the very same slot. Each of these undetected collisions will result in a collision of messages as soon as the schedule is executed[5].

Undetected collisions and unused slots are the two reasons why convergence of the schedule is crucial for the efficiency of the scheduling algorithm. The optimal configuration should thus be a combination of r and b that allows the schedule to converge in most of the cases.

Unfortunately, it is not possible in practice to determine whether a schedule has already converged, since all slots contain (collisions of) randomly chosen fingerprints that remain secret to their respective owner. In the simulation, however, all clients "play with an open deck"; the simulation can detect schedule convergence and can stop the scheduling process as soon as the schedule has converged. With this method, not more than the optimal number of rounds is executed. Let r_b^* be the number of rounds at

[5]See Chapter 4 for details on how collisions are handled in the prototype implementation.

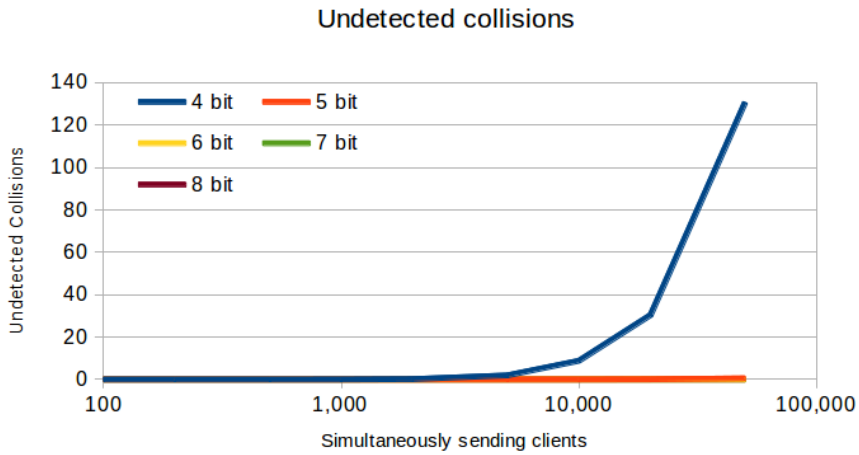

Figure 3.3: The number of undetected collisions after 32 rounds for various numbers of clients. The readings for 5 to 8 bit fingerprints all range between 0 and 0.5. See Table A.3 for exact measurement results.

which the schedule converges for a given b. Figure 3.4 shows the data volume profile with $r = r_b^*$ for different values of b. The results show that the performance of the algorithm benefits from a wide fingerprint range; 8 bit fingerprints naturally require more data to be transmitted than 6 bit fingerprints. But since the schedule converges faster when 8 bit fingerprints are used, r_8^* is lower than r_6^*, which causes the overall performance of an 8 bit fingerprint to outperform a 6 bit fingerprint.

We saw that, if r is chosen to match convergence perfectly, 8 bit fingerprints maximize the efficiency of the scheduling algorithm. In practice, the protocol can not stop as soon as the schedule converged. Therefore, the data volume profile can only be optimised by choosing a good estimation of r_8^* for the target network. Figure 3.5 shows the data volume profile for 8 bit fingerprints with different values for r. For comparison, the chart also shows the plot of the data volume profile for r_8^* (labelled CO). These results show that schedule convergence does not have to be achieved by all means; 16 rounds per cycle perform better than r_8^*, meaning that performance can be slightly increased by compromising possible collisions and empty slots. However, the plot for $r = 12$ shows that the data volume profile grows significantly as soon as r is chosen too small for the number of users on a network. Again, the benchmarks simulate a network in which all clients are constantly trying to send data.

The optimal configuration

In general, 8-bit fingerprints achieve the fastest schedule convergence. It depends on the size of the network how r should be chosen. For networks with up to $5,000$ users, 12 rounds per cycle maximize scheduling performance. For larger networks with up to $50,000$ users, 16 rounds per cycle are required to ensure sufficient schedule convergence.

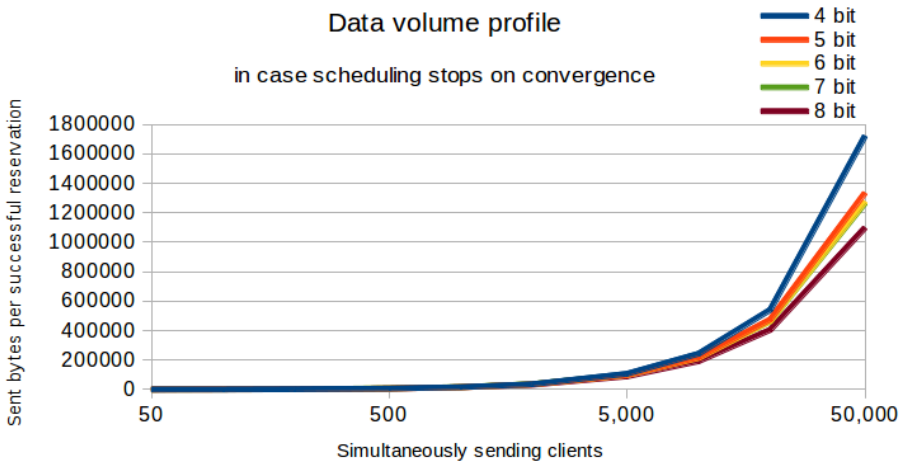

Figure 3.4: The data volume profile in case that the scheduling process stops as soon as the schedule converged. See Table A.4 for exact measurement results.

For the remainder of this chapter, Fingerprint scheduling will be used with 16 rounds per cycle and 8-bit fingerprints.

In order to judge the accomplishments of Fingerprint scheduling, the remainder of this chapter will describe other attempts to organize transmissions in DC-nets, and compare the performance of those attempts to the performance of Fingerprint scheduling.

3.3 Other scheduling protocols

In [2], Chaum describes two contention protocols that could be used to avoid collisions. The first one is better known as Slotted ALOHA protocol [10]: Users start their transmission whenever they want. In case of a collision, the user waits for a random period of time and then transmits the same message again. Transmissions are aligned to *slots* in time to decrease the number of possible collisions. This doubles the throughput rate, compared to pure ALOHA where transmissions are not aligned to fixed slots [13, p. 283]. Figure 3.6 illustrates how the protocol works.

Strictly speaking, this protocol does not even attempt to *schedule* transmissions. Nevertheless, it performs well when few clients try to transmit data simultaneously – it adds no organizational overhead and when a single client starts a transmission, they can use the full available bandwidth immediately. In large networks, however, there will be many clients that try to transmit data simultaneously. This surfaces a shortcoming of the protocol: With many transmitting clients, collisions might appear again and again, even if clients randomize the idle time between two transmission attempts. It is not guaranteed that a single transmission succeeds, which can cause the throughput to collapse. This protocol is therefore not suited to be used in large networks.

Chaum describes a second protocol which uses a reservation vector, with which users

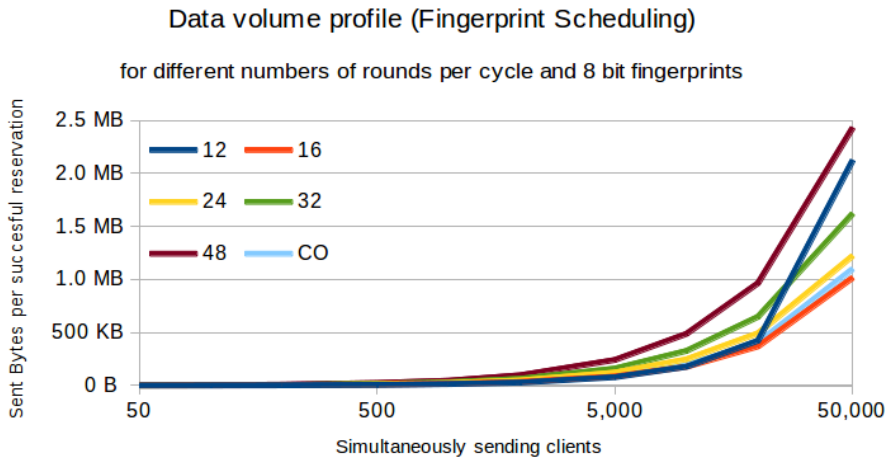

Figure 3.5: The data volume profile for different values of r. The plot labelled *CO* shows the data volume profile for r_8^*. See Table A.5 for exact measurement results.

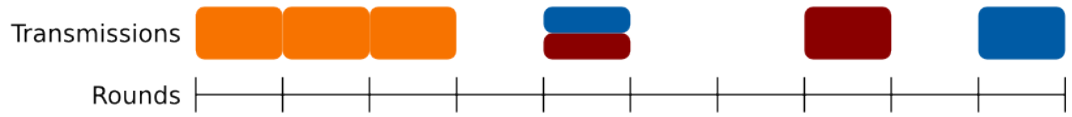

Figure 3.6: An illustration of the slotted ALOHA protocol. Blocks of different colours represent transmissions by different users. Multiple blocks in a single round represent a collision.

collectively create a schedule. This protocol consist of a reservation round and several execution rounds where the schedule is put into action. In the reservation round, each client may revert one random bit of the reservation vector. The outcome of this round determines the schedule for the second phase: The client that inverted bit i of the reservation vector can use round i of the execution phase to send a message. Figure 3.7 illustrates how this protocol works. The DC-net implementation Herbivore ([5]) uses this scheduling protocol. The implementation will further be discussed in Chapter 6.

This protocol only operates well with a scheduling vector that fits the number of transmitting clients. If the scheduling vector is chosen too small, then chances are that multiple clients try to reserve the same slot. They all invert the corresponding bit and those bits will collide when the outcome of the round is calculated. When an even number of bits collide, they will all cancel out. In such a case the involved clients can detect the collision. An odd number of bits will result in a 1. All clients will assume that their reservation succeeded, so that their messages will collide when the schedule is put into practice. Both variants can lead to a throughput collapse since it is not guaranteed that a single transmission succeeds. By choosing a large reservation vector, the chance

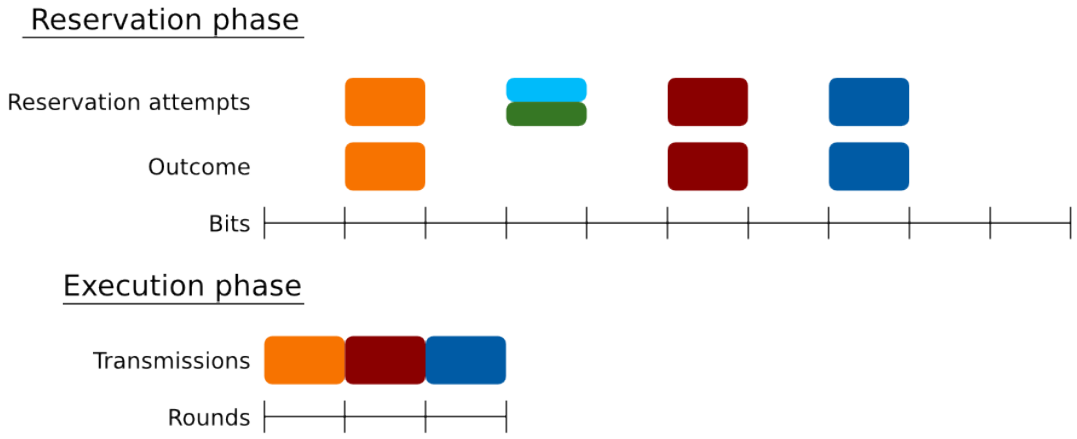

Figure 3.7: An illustration of the scheduling protocol used by Herbivore. Note that the bright-blue and green reservation attempts cancel out. Also, since each client knows how many slots are reserved, unreserved rounds can simply be skipped. Therefore the execution phase lasts for only three rounds, after which the next scheduling phase is started.

of colliding reservations can be decreased so that a throughput collapse is unlikely. The analysis in [5] shows that the best performance is achieved with a scheduling vector of size "$m_r = k_r\sqrt{p}$, where p is the packet size, and k_r is the approximate number of nodes that will transmit in a given round" [5, p. 9f].

The next section will compare the performance of the described scheduling algorithms. These measurements will show that Herbivore's protocol can remain stable, even if thousands of users try to transmit simultaneously. However, it produces much more overhead than other scheduling algorithms, including Fingerprint scheduling.

A third scheduling protocol is based on the work of Pfitzmann. In [7, p. 188ff], he presents an efficient algorithm to detect collisions. Instead of using a scheduling vector, each client repetitively transmits a packet that holds the index of the slot that the client tries to reserve, together with a 1 that is used to determine the number of active clients. The content of the packets are not combined using a binary XOR operation. Instead, they are summed up. The outcome of the round will thus contain two values: The sum s of all slots that clients try to reserve, and c, the number of clients that try to reserve a slot. From that, each client calculates $\varnothing = \lfloor s/c \rfloor$. In the next round, all clients that are trying to reserve a slot with an index lower than $\varnothing$ will transmit their packet again. This narrows down the number of transmitting clients round by round. Eventually, there are two possible outcomes. If the outcome of two subsequent rounds is the same, then all involved clients are trying to reserve the same slot. Their reservation attempt failed at this point. Otherwise, if only one client transmits a packet, then the reservation attempt of this client is successful. The value $\varnothing$ is used to recursively branch off the algorithm, as illustrated in Figure 3.8.

The performance of this algorithm is influenced by three parameters: The number

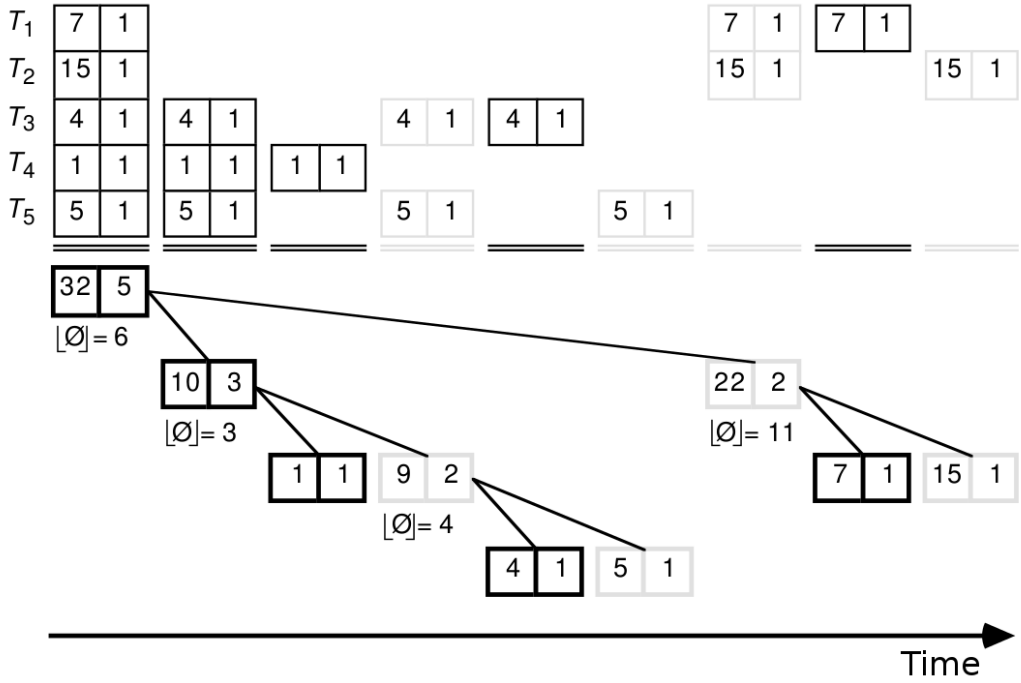

Figure 3.8: Picture is taken from [7, p. 189]. Each row in the upper part of the picture shows what the stations T_1 through T_5 send. The sum of all packets can be found in the lower part of the picture, together with the average value $\varnothing$ that was transmitted in each round. A station transmits its packet again whenever it tries to reserve a round that has an index lower than the current average value. The content of grey boxes is not transmitted, it can locally be deduced from earlier transmissions.

of clients, the number of slots that each client may attempt to reserve at the same time, and finally the overall number of slots that a client can choose from. The number of clients is given by the network that the algorithm is operated in. The number of parallel reservations should be limited to 1 in big networks to avoid unfair distribution of reservations. The number of slots heavily influences the chance of collisions, but unfortunately Pfitzmann does not state how this parameter should be chosen. Appendix A.1 describes how to optimize this parameter.

3.4 Performance Comparison

This section compares the performance of Fingerprint scheduling to the performance of Pfitzmann's algorithm as well as the algorithm used in Herbivore. Since all algorithms work differently, the performance will be compared based on one single statistic. This statistic will be what was called *data volume profile* in the previous section: The amount of data that a client has to send in order to successfully reserve a slot.

Similar to the measurements in Section 3.2, the performance of the different scheduling algorithms have been measured with simulations.

Figure 3.9 shows a comparison of the performance of all three scheduling algorithms. Unsurprisingly, Herbivore does not perform as good as Pfitzmann's scheduling algorithm or Fingerprint scheduling, due to the fact that it is not designed for networks of this large a size. Pfitzmann's algorithm, however, operates at a low data volume profile, even in large networks. Fingerprint scheduling requires approximately five times more data to be transmitted than Pfitzmann's algorithm.

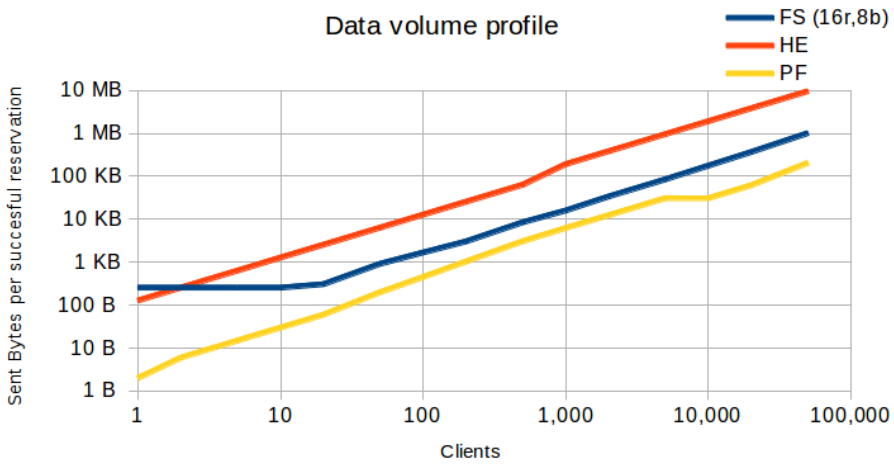

Figure 3.9: The data volume profile of all three scheduling algorithms. *PF* for Pfitzmann's algorithm, *HE* for the algorithm that is used for Herbivore, and *FS* for Fingerprint scheduling. See Table A.6 for exact measurement results.

Despite the fact that Fingerprint scheduling has a higher data volume profile than Pfitzmann's algorithm, there are reasons why Fingerprint scheduling is the scheduling algorithm of choice for large networks. When Pfitzmann's algorithm is used, stations must not drop out in between. If a station drops out in the middle of the scheduling protocol, then the whole protocol breaks and can not be completed. For networks with n sending stations, this means that all sending stations must be available for about n continuous rounds. For $50,000$ users, this would be about 350MB of data[6]. In a network where all clients are reliably connected over a long period of time, the efficiency of the network can benefit from the low data volume profile of Pfitzmann's algorithm. But in case that connections are unreliable or stations leave the network at unpredictable times, it might rarely be possible to complete the protocol of Pfitzmann's algorithm.

Fingerprint scheduling is resistant against stations that drop out. The particular round in which a station drops out has to be repeated, but besides that, the protocol can continue normally. In case that a slot was reserved by the station that dropped out,

[6]The packet size was calculated following Pfitzmann's proposed packet layout in [7, p. 188]

this slot will appear free in all following rounds, allowing other stations to shift towards this slot. This property makes Fingerprint scheduling well-suited for networks where connections can not be assumed to be stable and clients join or leave rapidly.

3.4.1 Client activity

While being connected to a network, clients do not necessarily want to send data constantly. A client will be called *active* in the periods in which they want to send data. Depending on the application, clients might only be active for a small fraction of the time that they are connected to the network. This section will demonstrate the influence of client activity on the data volume profile.

Figure 3.10 shows the data volume profile of all three scheduling algorithms for different values of client activity rates. With a client activity rate of 10%, each client will attempt to reserve a slot in one out of ten opportunities.

The data volume profile of Pfitzmann's scheduling algorithm is not affected by the change in activity – its data volume profile is only influenced by the number of clients that are connected, whether or not they are active. However, the data volume profiles of Herbivore and Fingerprint scheduling do scale with respect to client activity. Fingerprint scheduling naturally scales with respect to client activity – no further estimations are necessary. Herbivore achieves this by estimating the activity rate ([5, p. 9f]). The measurement results show the data volume profile for the case that this estimation is perfect. The performance of Pfitzmann's algorithm can possibly be improved with a similar estimation of client activity.

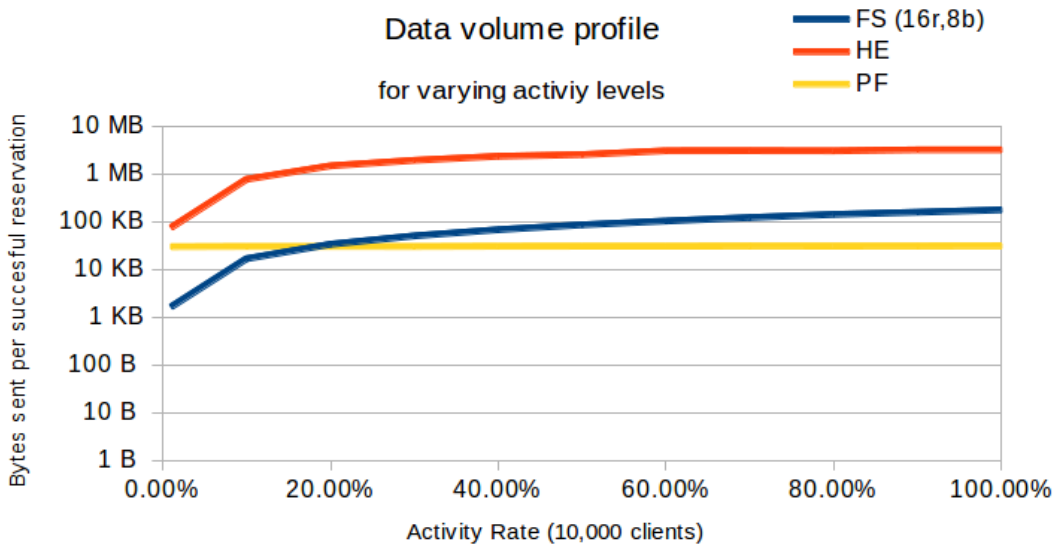

Figure 3.10: The data volume profile of all three scheduling algorithms, given different client activity rates. See Table A.7 for exact measurement results.

As a novel scheduling algorithm that is designed for large networks, the Fingerprint scheduling protocol achieves efficient scheduling at a low data volume profile. Pfitzmann's algorithm requires even less data to be transmitted, but in contrast to Pfitzmann's algorithm, Fingerprint scheduling is resistant against stations that drop out throughout the execution of the protocol. This difference makes Fingerprint scheduling in particular well-suited to be used in situations where reliable connectivity can not be guaranteed.

Chapter 4

Implementation details

This chapter will give some more information about the prototype implementation, as well as the results of measurements concerning bandwidth utilization.

The prototype implementatino is written in Java and requires Java 6 or newer to be run. The implementation uses native Java network sockets to send and receive data. Out of the box, these sockets will perform poorly when small packets are used. The socket will not send a packet right away, but will store the packet in a buffer, for the case that it can be combined with other packets to reduce overhead. This buffering technique creates a delay that significantly lowers the network throughput. When tested on a local network with 1024 byte packets, the server and client application sent 300kByte/s. Without this buffering technique, they achieve a throughput of 2MByte/s.

4.1 Message handling

This section briefly describes what happens when a user wants to send data. In a network with a lot of active participants, clients can not send messages as soon as the user enters them. It depends on the outcome of the scheduling protocol in which round a station is allowed to transmit. Therefore, messages are stored in a variably sized buffer with a first-in first-out policy, which ensures that messages are not delivered out of order.

If a station is allowed to send in a certain round, it processes the content of the message buffer as illustrated in Figure 4.1. First, the head of the message buffer is stored as *Pending Message*. In the next step, 10-padding is applied, as described in Chapter 2. The result forms the message payload of the composed packet. Note that the packet header is omitted in the illustration, as well as scheduling data. *Packet capacity* describes the capacity of the section that is purely designated to message payload.

At the end of a round, the station checks whether the outcome of the round equals the message that this station transmitted. If this is not the case, a collision has occurred and the pending message remains at the head of the buffer. This way it will be retransmitted the next time this station sends a message, otherwise it is removed from the buffer.

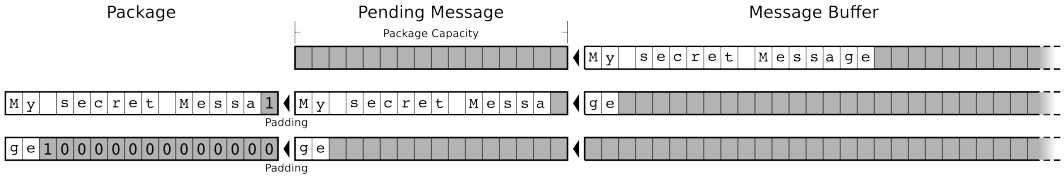

Figure 4.1: The transitions between the message buffer and actual packages. See Chapter 2 for details on how 10-padding is applied.

4.2 Throughput benchmarks

Some tests were made to see if the prototype implementation is able to use the full available bandwidth. The tests were run on a local network, using two computers. One computer simulates multiple clients, while the other computer acts as a server. The computers are connected via CAT6 Ethernet cables and through a 100Mbit switch. Both computers are equipped with a 100Mbit Ethernet port. Fingerprint scheduling is used with 8 bit fingerprints and 16 rounds per cycle. Table 4.1 shows the measurement results. Due to the nature of Fingerprint scheduling, three clients can each only reserve one out of 16 slots per cycle. As a consequence, in both measurements each client was able to send one packet per cycle. This shows that the network bandwidth forms the bottleneck of the throughput: When all 16 clients share the available bandwidth, then the individual throughput sinks to less than a third compared to the individual throughput that each client achieves when only three clients have to share the available transmission bandwidth.

	Transmission time	Individual throughput	Overall throughput
3 clients	3.78 seconds	12.69 KByte / sec	1.0 MByte / sec
16 clients	9.53 seconds	5.33 KByte / sec	1.6 MByte / sec

Table 4.1: Transmission benchmarks on a local network, for transferring files of 50 KByte, using 1 KByte packets.

Chapter 5

Possible improvements

The software that was written for this thesis is a prototype implementation. Although it comes with a full command-line interface[1], it would definitely need some polishing and refinements before it could be used in production. It can, however, be used for further research on large-scale DC-networks. Specifically, the *fail-stop* broadcast protocol presented in [14] could be implemented. Attackers can reveal information about recipients by letting some stations on the network receive a different outcome of a round than the rest of the network. The fail-stop protocol can help to discover this so-called inconsistent broadcast.

The remainder of this chapter will address two other interesting possible improvements.

Erlang

The provided implementation is written in Java in order to have a big shared code base between the server and the client application. This helped to speed up the development process. An actual implementation, however, that is supposed to be used in production, could possibly benefit from another programming language. A DC-net server has to handle many connections simultaneously. All connections produce a constant, high, and symmetrical load. Furthermore, many calculations are concurrency-sensitive. The prototype implementation shows that it is indeed possible to base a DC-net server on Java and maintain maximal bandwidth utilization for networks of 30 clients. But Erlang might be the language of choice when it comes to networks with thousands of users.

Erlang[2] is a functional programming language, designed with heavy emphasis on concurrency. The messaging service WhatsApp[3] uses Erlang on their servers to handle message traffic. In 2012, WhatsApp claimed [9] to have achieved a throughput of more than 200,000 distinct messages per second on a single machine, handling more than 2 million TCP connections at a time.

[1] See Appendix B for an overview of the available commands.

[2] `http://en.wikipedia.org/wiki/Erlang_(programming_language)`

[3] `http://en.wikipedia.org/wiki/WhatsApp`, a messaging client

This application shows that Erlang is a promising programming language to be used as a back-end of a DC-net server, since the load that a DC-net server has to handle is comparable to the load that a messaging server is exposed to. In both cases, the traffic load is symmetric – each user sends as much to the server as the server sends back to the user. In WhatsApp's use case, the incoming and outgoing packets might not target the same user, but the traffic balance stays the same. On top of that, both applications require a loss-free transmission technique, where other traffic-intensive web services like YouTube[4] or Twitch[5] can drop back to loss-tolerant transmissions.

Since Erlang is well-suited for concurrency and parallelism, it is worth testing if an Erlang port could lower the workload of a DC-server.

Multicast

The client-server structure that is used in this implementation forms an interesting application for multicast transmissions. At the end of a round, the server broadcasts the outcome of the round to all n connected stations. In the current implementation, the server does so by sending one packet to each station separately, although the content of all packets is exactly the same. Multicast transmissions can help to make this step more efficient. In contrast to unicast, multicast describes a transmission that has one sender and multiple recipients, where unicast has only a single recipient. For this application, the server could send a single packet that is addressed to all n clients, instead of sending n packets to n stations. This could potentially lower the outgoing network load significantly.

Multicast is mostly used in cases where data loss is not an issue (e.g. streaming a video recording of a live event to thousands of users simultaneously). In our case, however, multicast would be applied when the server transmits the outcome of a round back to the clients. This requires a protocol that ensures a reliable data transmission. *Pragmatic General Multicast*[6] is a multicast protocol that has this property. It is worth investigating if the network throughput could be further improved by using this protocol.

[4] `http://en.wikipedia.org/wiki/Youtube`, a widely used video platform

[5] `http://en.wikipedia.org/wiki/Twitch_(website)`, a platform that offers live-stream broadcasting

[6] [RFC 3208] Speakman, et. al., "PGM Reliable Transport Protocol Specification", December 2001, `http://tools.ietf.org/html/rfc3208`

Chapter 6

Related work

DC-nets have been implemented before. In this chapter, we will briefly give an overview of relevant existing implementations and describe their properties. First of all, Pfitzmann's extensive work on DC-nets should be mentioned, which goes far beyond the protocol described in Section 3.3.

Herbivore

Herbivore [5] is a DC-net implementation that was developed by Goel et al. in 2003. It has some characteristic features that make it scalable with respect to the number of participants without compromising throughput; instead of having a single network that contains all users, Herbivore automatically splits the set of users into subsets of equal size. These subsets are called cliques. Each of these cliques then forms a separate DC-net. Different mechanics prevent denial of service attacks as well as attacks on sender and receiver anonymity:

- Users can not choose which clique they are part of. This ensures that a clique can not be flooded by malicious participants, which would weaken the anonymity of all remaining participants of that clique. Users can, however, join a different clique. This can help to escape cliques that are blocked by malicious participants. Because of the way the clique-selection mechanism works, an attacker can not feasibly follow a user to a different node [5, p. 12].

- In small networks where participants frequently join and leave, the sender anonymity can not be guaranteed for long-lasting transmissions. The sender of a long-lasting transmission can only be a client who was continuously connected to the network for the entire duration of the transmission. Depending on the behaviour of other clients, this might allow an attacker to shrink the set of possible senders of such a long-lasting transmission. Clients can (anonymously) announce that they wish other stations to stay connected in order to protect their identity until the long-lasting transmission has been finished. Furthermore, Herbivore prevents clients

from continuing transmissions as soon as their identity could be revealed otherwise [5, p. 7, 12].

Herbivore is designed to perform best in small cliques with less than 40 participants, but the size of the overall network can be much higher. The scheduling algorithm which is used by this implementation is discussed in Section 3.3.

Dissent

Another noteworthy, sophisticated implementation, called Dissent [3], was created by Ford and Corrigian-Ribs in 2010. The initial publication is intended to be used by a small, closed group of participants. Nevertheless, this implementation does offer methods to trace down the source of a DoS attack, which overcomes a natural vulnerability of native DC-nets. At the same time, this implementation does not target interactive application. Initially, Dissent focussed on resistance against attacks, rather than on efficiency.

A more recent publication in 2012 [15], however, shows that Dissent can be used in networks with up to $5,000$ user, with a latency of 600ms. Even in networks of this size, it is still possible to trace down DoS attackers. This publication features a client-server structure in order to distribute computation and network load. Also, clients only share secret keys with servers, rather than with other clients. This design decision has two important advantages: It lowers the computation load that a client has to handle, and, more importantly, it makes it unnecessary to repeat a round as soon as a single client disconnects from the network [15, p. 5f].

Chapter 7

Conclusion

The design of the prototype implementation that was described in Chapter 2 combines several ideas on how to overcome the challenges of large-scale DC-networks in practice. The prototype implementation itself can be used in future research on large-scale DC-networks. Examples for possible extensions have been given in Chapter 5. Furthermore, the prototype implementation demonstrates the usage of the novel scheduling algorithm that was introduced in Chapter 3. This algorithm improves existing scheduling methods as it offers efficient scheduling, even for large DC-networks, while remaining stable despite unreliable client connectivity.

Chapter 8

Acknowledgements

I would like to thank Anna Krasnova and Peter Schwabe[1] for their constructive feedback and motivating support, even when my focus shifted away from the original plans. Without this freedom, Fingerprint scheduling would not have left the draft status.

Finally, I would like to thank my parents for supporting me, even during the times when my day-night cycle turned into that of a vampire.

[1] in no particular order

Bibliography

[1] Daniel J. Bernstein. The salsa20 family of stream ciphers. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer Berlin Heidelberg, 2008. http://dx.doi.org/10.1007/978-3-540-68351-3_8.

[2] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988. http://dx.doi.org/10.1007/BF00206326.

[3] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 340–350, New York, NY, USA, 2010. ACM. http://doi.acm.org/10.1145/1866307.1866346.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, pages 22–38, 2004. https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf.

[5] Sharad Goel, Mark Robson, Milo Polte, and Emin Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003. http://www.cs.cornell.edu/People/egs/herbivore/herbivore.pdf.

[6] Anna Krasnova. Analysis and enhancement of existing approaches towards an implementation of a dc-net, 2014.

[7] Andreas Pfitzmann. *Datensicherheit und Kryptographie*. PhD thesis, 1997.

[8] Andreas Pfitzmann and Marit Hansen. Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management-a consolidated proposal for terminology. *Version v0*, 31:15, 2008.

[9] Rick Reed. Scaling to millions of simultaneous connections, 2012. http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf.

[10] Lawrence G. Roberts. Extensions of packet communication technology to a hand held personal terminal. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 295–298, New York, NY, USA, 1972. ACM. http://doi.acm.org/10.1145/1478873.1478910.

[11] Andrei Serjantov, Roger Dingledine, and Paul Syverson. From a trickle to a flood: Active attacks on several mix types. In *Information Hiding*, pages 36–52. Springer, 2003. http://www.dtic.mil/dtic/tr/fulltext/u2/a465475.pdf.

[12] Georgos Siganos, Sudhir Leslie Tauro, and Michalis Faloutsos. Jellyfish: A conceptual model for the as internet topology. *Communications and Networks, Journal of*, 8(3):339–350, Sept 2006. http://intersci.ss.uci.edu/~drwhite/pw/Foulatsos_paper.pdf.

[13] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Press, Upper Saddle River, NJ, USA, 5th edition, 2010.

[14] Michael Waidner. Unconditional sender and recipient untraceability in spite of active attacks. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, volume 434 of *Lecture Notes in Computer Science*, pages 302–319. Springer Berlin Heidelberg, 1990. http://dx.doi.org/10.1007/3-540-46885-4_32.

[15] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. http://dedis.cs.yale.edu/dissent/papers/osdi12.pdf.

Appendix A

Details on measurement results

This chapter will describe how exactly the various performance measurements in Chapter 3 were obtained. Section A.2 will also give the exact measurement results. As stated earlier, simulations of the algorithms were used instead of measuring the performance in actual DC-networks. The source code of those simulations can be found in the package `benchmarking`. The benchmarks can also be executed from the command line. A full overview of the available commands is given in Appendix B.

The simulation of Fingerprint scheduling follows exactly the protocol that was presented in Section 3.4. The simulation of Pfitzmann's algorithm is modelled according to his description in [7]. This simulation distinguishes between messages that have to be transmitted and packets that can be deduced from earlier transmissions, so that the calculated data volume profile of the simulation matches precisely the real amount of data that would be necessary to resolve the schedule. The data volume profile of all three simulations are rounded up to the next byte, since bitwise transmissions are not practical. The size of transmission overhead that is added by underlying communication protocols is not taken into account, as this depends on the implementation.

A.1 Configuring Pfitzmann's algorithm

As stated in Section 3.4, Pfitzmann's scheduling algorithm is configured by three parameters: The number of slots in the schedule (B), the number of clients (s), and finally the number of slots that a client may reserve in each scheduling cycle (m). This section will briefly explain which values were used for those parameters in the simulations, and why they were chosen.

The number of clients is naturally given by the network that the protocol is run in. The number of slots that may be reserved is partly given by the size of the network – in a network with potentially thousands of users, it would not be reasonable to allow clients to reserve more than one slot at a time. Building up a schedule of this size is hard enough on its own. Having clients reserve multiple slots at once would not help to ease this task. The number of slots in the schedule is not externally defined. Instead, B should be chosen according to the size of the network. The more users are on the network, the

more collisions can presumably occur in the schedule. In order to avoid those collisions, B should be chosen much higher than the number of users on the network. Assuming that clients choose random slots in the schedule, the collision probability underlies the birthday paradox, just as the scheduling algorithm used in the Herbivore implementation (see Section 3.4). Just as proposed in [5], the size of the scheduling vector depends on the number of clients that are on the network. Let i be the ratio between B and s, so that $B = i * s$. Figure A.1 gives an overview of the data volume profile for different values of i. Pfitzmann's algorithm achieves the minimal data volume profile for $i = 32$. Based on these results, the simulations were executed using $i = 32$.

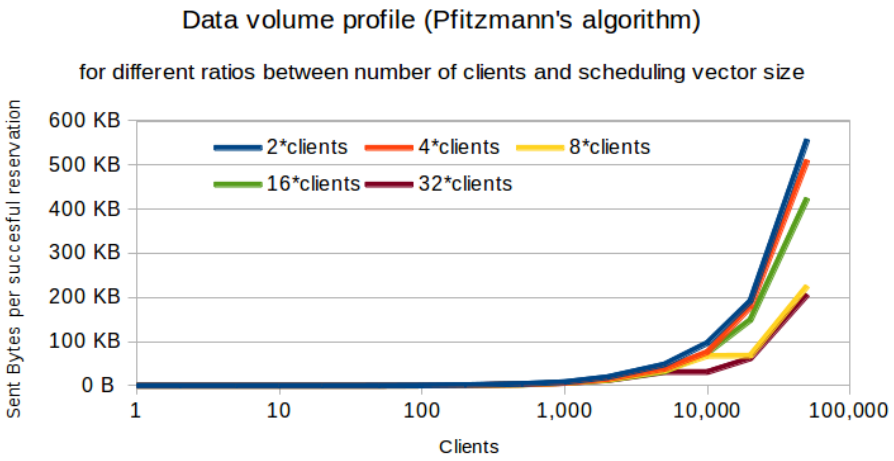

Figure A.1: The data volume profile for Pfitzmann's algorithm, using different ratios between the number of users and the number of slots. See Table A.8 for exact measurement results.

A.2 Exact measurement results

Section 3.2 through 3.4 contains various plots, showing measurement results that were obtained with simulations. This chapter will list the exact measurement results for each test that was made, along with the commands that can be used on the command line interface in order to reproduce the test.

Clients	50	100	200	500	1,000	2,000	5,000	10,000	20,000	50,000
4 bit	4.00	14.00	20.07	24.92	30.02	36.25	42.44	47.99	53.26	59.54
5 bit	4.00	9.50	16.17	21.76	25.09	27.55	31.59	34.73	38.01	42.19
6 bit	9.40	11.93	14.26	16.28	17.37	18.95	21.22	22.80	24.33	26.45
7 bit	9.25	10.84	12.00	14.40	15.73	17.26	19.00	20.44	21.58	23.39
8 bit	8.95	11.16	12.05	13.23	14.86	16.55	17.22	18.98	19.63	21.23

Table A.1: The number of required rounds in relation to the number of used bits for fingerprints. Figure: 3.1. Command: `run scripts/benchmarks/fingerprint_required_rounds`

Clients	50	100	200	500	1,000	2,000	5,000	10,000	20,000	50,000
4 bit	0.34	0.37	0.38	0.34	0.48	0.88	1.71	2.54	3.08	3.52
5 bit	0.20	0.42	0.36	0.33	0.30	0.35	0.46	0.67	1.02	1.77
6 bit	0.40	0.39	0.38	0.37	0.38	0.38	0.39	0.39	0.43	0.53
7 bit	0.50	0.32	0.30	0.40	0.36	0.42	0.36	0.40	0.37	0.40
8 bit	0.38	0.41	0.41	0.41	0.42	0.41	0.50	0.53	0.33	0.36

Table A.2: The number of empty slots after 32 rounds in relation to the number of used bits for fingerprints. Figure: 3.2. Command: `run scripts/benchmarks/fingerprint_size`

Clients	1,000	2,000	5,000	10,000	20,000	50,000
4 bit	0.0186	0.1737	1.9383	8.7885	30.3759	130.6563
5 bit	0	0	0.0013	0.0197	0.08247	0.5393
6 bit	0	0	0	0	0.0001	0.0010
7 bit	0	0	0	0	0	0
8 bit	0	0	0	0	0	0

Table A.3: The number of undetected collisions after 32 rounds in relation to the number of used bits for fingerprints. Figure: 3.3. Command: `run scripts/benchmarks/fingerprint_size`

Clients	50	100	200	500	1,000
4 bit	128.00 B	687.68 B	1.99 KB	6.64 KB	15.74 KB
5 bit	160.00 B	817.60 B	2.10 KB	6.87 KB	15.62 KB
6 bit	144.00 B	846.72 B	2.20 KB	7.09 KB	15.51 KB
7 bit	280.00 B	1.27 KB	2.44 KB	7.92 KB	16.67 KB
8 bit	392.96 B	1.21 KB	2.44 KB	6.37 KB	15.85 KB
Clients	2,000	5,000	10,000	20,000	50,000
4 bit	36.11 KB	107.22 KB	243.35 KB	541.16 KB	1.72 MB
5 bit	34.72 KB	99.46 KB	218.74 KB	475.81 KB	1.33 MB
6 bit	36.18 KB	97.83 KB	214.63 KB	464.17 KB	1.28 MB
7 bit	37.10 KB	100.81 KB	219.95 KB	463.49 KB	1.27 MB
8 bit	32.87 KB	89.38 KB	192.55 KB	405.13 KB	1.10 MB

Table A.4: The data volume profile in case that the scheduling protocol stops on convergence. Figure: 3.4. Command: `run scripts/benchmarks/fingerprint_convergence`

Clients	50	100	200	500	1,000
12	720.00 B	1.14 KB	2.58 KB	6.30 KB	13.38 KB
16	921.60 B	1.68 KB	3.05 KB	8.51 KB	15.99 KB
24	1.21 KB	2.64 KB	4.61 KB	11.78 KB	24.26 KB
32	1.62 KB	3.23 KB	6.32 KB	15.90 KB	31.86 KB
48	2.44 KB	5.00 KB	9.45 KB	24.65 KB	46.76 KB
CO	392.96 B	1.21 KB	2.44 KB	6.37 KB	15.85 KB
Clients	2,000	5,000	10,000	20,000	50,000
12	27.90 KB	78.69 KB	177.13 KB	422.35 KB	2.12 MB
16	33.93 KB	84.98 KB	176.95 KB	368.60 KB	1.02 MB
24	48.93 KB	122.80 KB	246.98 KB	492.33 KB	1.22 MB
32	64.32 KB	160.60 KB	326.46 KB	648.10 KB	1.62 MB
48	97.69 KB	241.20 KB	487.21 KB	964.51 KB	2.43 MB
CO	32.87 KB	89.38 KB	192.55 KB	405.13 KB	1.10 MB

Table A.5: The data volume profile for different numbers of rounds per cycle. Figure: 3.5. Command: `run scripts/benchmarks/fingerprint_rounds`

Clients	50	100	200	500	1,000
FS (16r,8b)	921.60 B	1.68 KB	3.05 KB	8.51 KB	15.99 KB
HE	6.40 KB	12.80 KB	25.60 KB	64.00 KB	192.03 KB
PF	200.00 B	452.00 B	1.04 KB	3.08 KB	6.26 KB
Clients	2,000	5,000	10,000	20,000	50,000
FS (16r,8b)	33.93 KB	84.98 KB	176.95 KB	368.60 KB	1.02 MB
HE	384.03 KB	960.00 KB	1.92 MB	3.84 MB	9.60 MB
PF	12.44 KB	30.99 KB	30.85 KB	62.11 KB	206.34 KB

Table A.6: The data volume profile for three different scheduling algorithms. Figure: 3.9. Command: `run scripts/benchmarks/data_volume_profile`

Activity	1%	10%	20%	30%	40%	50%
FS (16r,8b)	1.63 KB	16.76 KB	33.69 KB	50.46 KB	67.87 KB	85.92 KB
HE	75.00 KB	774.94 KB	1.48 MB	1.94 MB	2.36 MB	2.56 MB
PF	29.85 KB	30.14 KB	30.22 KB	30.11 KB	30.52 KB	30.55 KB
Activity	60%	70%	80%	90%	100%	
FS (16r,8b)	103.50 KB	122.18 KB	142.21 KB	157.73 KB	176.61 KB	
HE	3.08 MB	3.08 MB	3.07 MB	3.22 MB	3.22 MB	
PF	30.60 KB	30.73 KB	30.72 KB	30.80 KB	30.92 KB	

Table A.7: The data volume profile for the three scheduling algorithms, in relation to client activity. Figure: 3.10. Command: `run scripts/benchmarks/activity`

Clients	50	100	200	500	1,000
2*clients	237.06 B	594.80 B	1.62 KB	4.08 KB	8.09 KB
4*clients	161.76 B	478.28 B	1.29 KB	3.05 KB	6.38 KB
8*clients	208.00 B	464.00 B	1.13 KB	2.75 KB	6.88 KB
16*clients	216.00 B	428.00 B	1.06 KB	2.67 KB	6.53 KB
32*clients	208.00 B	416.00 B	1.03 KB	3.10 KB	6.15 KB
Clients	2,000	5,000	10,000	20,000	50,000
2*clients	19.39 KB	48.14 KB	96.87 KB	191.61 KB	557.59 KB
4*clients	15.49 KB	38.15 KB	76.15 KB	178.80 KB	510.82 KB
8*clients	13.66 KB	34.01 KB	67.78 KB	68.18 KB	225.53 KB
16*clients	12.78 KB	32.09 KB	74.39 KB	148.92 KB	424.44 KB
32*clients	12.44 KB	30.93 KB	30.97 KB	61.88 KB	206.11 KB

Table A.8: The data volume profile of Pfitzmann's algorithm for different ratios between the number of slots and clients. Figure: A.1. Command: `run scripts/benchmarks/pfitzmann_ratio`

Appendix B

Command line interface reference

This chapter contains an overview of all available commands along with a short explanation on how to use them. Again, the source code of the prototype implementation can be found at `https://github.com/25A0/DCnet/archive/v0.3.zip`.

B.1 Build and run

In order to build the program from source, run `make` on Linux or MacOS or manually run `javac -d bin/ -cp src/ src/component/Main.java`. This requires Java 6 SDK or a newer version of Java. Once the build has completed, run `dcnet.sh` or manually run `java -cp bin/ component.Main`. You can pass the path to one or multiple scripts as arguments that will be executed automatically. The folder `scripts/` contains a couple of example scripts.

B.2 Basic program flow control

A line that starts with a `#` is treated as a comment.

```
# This is a comment
```

To quit the program, use

```
exit
```

Scripts can be used to automate operations. A script is a simple text file. All commands that can directly be called from the command line, can also be called through a script. This includes calling a script. Use the following command to run a script, where `<path>` points to the script relative to the directory from which the application is run. For example, `run scripts/benchmarks/activity` will run the benchmark scripts that reproduces the measurements presented in Figure 3.10.

```
run <path>...
```

B.3 DC-network commands

Use the following command to create a new station. The two variations `server` and `client` create a server or a client station, respectively. The optional parameter `-l | --local` creates a local server that can not be reached via network (for testing purposes). Specify a port in order to start a non-local server.

```
dc make ( client | server (-l | --local | <port>))
```

In order to obtain a list of all locally created stations, use:

```
dc list
```

Shared keys can be added with the following command:

```
dc <alias> keys add <foreign_alias> [(-k|--key) <key>]
```

Keys are 32 character strings, enclosed by quotation marks. If no key is provided, then a key will be automatically generated from the aliases of the stations. Note that this must only be used for testing purposes, since the keys are predictable.

In order to connect a station to a remote or local server, use the first or second command, respectively:

```
dc <alias> connect <address>:<port>
dc <alias> connect (-l | --local) <server-alias>
```

Note that `<alias>` can be the alias of a client or a server, which allows to connect servers to other servers. `<address>` is an IPv4 address.

As described in Chapter 2, the prototype implementation uses a two-stage mechanism in order to let stations join a network. Therefore, the following command is used to change the state of a station once it is connected to a network, or to query the current state of the station, respectively:

```
dc <alias> state (active | inactive)
dc <alias> state
```

Switching to `active` will cause the station to join the network, so that it can send messages. Switching to `inactive` will cause the station to leave the network again. In both modes, the station will be able to receive messages that are broadcast by the server that the station is connected to.

In order to send messages, use the following command to send a simple string message, or the content of a file, respectively:

```
dc <alias> send <message>
dc <alias> send (-f | --file) <path>
```

Note that the message has to be enclosed by quotation marks.

In order to read received messages, use the following commands to show all messages that a station has received since the last call of this command, or to write them to a file, respectively:

```
dc <alias> read
dc <alias> read (-f | --file) <path>
```

B.4 Debugging commands

Multiple levels of debug output can be displayed. Use the following command to change the level of detail, where `0` disables all debug information and `2` will show all available debug information.

```
debug level (0, 1, 2)
```

Certain *tagged* debug information can be tracked explicitly. This can be useful for monitoring scheduling procedures or changes in the network status. The following command prints a list of available tags. Tags are collected in runtime, the content of this list can therefore change sooner or later.

```
debug track list
```

The following command can be used to start or stop tracking a certain tag:

```
debug track (add | remove) <tag>
```

B.5 Scheduling benchmarks

The following commands help to repeat the measurements that were presented and analysed in Chapter 3. There are separate commands to run each of the discussed scheduling algorithms in a simulation. The command for fingerprint scheduling requires the number of slots and the number of bits per fingerprint. Optionally, a list of users can be provided. A list is enclosed by square brackets and its items are separated by whitespace characters. Example: `[1 2 5 10 100 1000]`. Other optional parameters are `<activity>`, which is a `double` value that sets the activity rate (default: 1.0), as well as a boolean value `<soc>`, which determines whether the scheduling protocol terminates as soon as the schedule converged (default: `false`).

```
scheduling fingerprint <slots> <bits> [<users>] [<activity> [<soc>]]
```

The command to test Pfitzmann's scheduling algorithm requires the ratio between the number of users and the number of available slots. A ratio of 32 means that there are 32 slots available per user. The number of users can be given as a list, as described above. Finally, the activity rate can be specified as a `double` value.

```
scheduling pfitzmann <ratio> [<users>] [<activity>]
```

The test for the scheduling algorithm used in Herbivore does not necessarily require any parameters. Optional parameters are the number of users as a list, and the activity rate, as described above.

```
scheduling herbivore [<users>] [activity]
```