Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Jan Müller

Protecting Blockchain Applications with Programmable Networks

Semester Thesis SA-2018-15
March 2018 to June 2018

Tutor: Prof. Dr. Laurent Vanbever
Supervisor: Maria Apostolaki

Abstract

Bitcoin is the worlds largest cryptocurrency and is of high interest by research and financial communities and by many individuals around the world. Its fame led to the discovery of various attacks against the bitcoin ecosystem. This thesis contributes to a framework which protects the bitcoin network against routing based isolation attacks. For this, we propose a modified bitcoin client for the use with a switch relay network and show that this new design is able to support multiple thousands of concurrent connections.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When the bitcoin protocol was designed in 2008, nobody could have imagined that it would become an internationally accepted currency of the size it is today. 10 years later, with a marked cap of 237 billion dollars and 200'000 daily transactions (as of 1.1.18) [1], it is still the largest crypto currency in the world. In the past few years, the term bitcoin has become a synonym for cryptocurrencies. Today, not only technically-versed people but also many individuals around the world are owning bitcoins or another cryptocurrency. This fast growth also drew the attention of the research community worldwide. The goal is to keep this global scale experiment of an authority free currency secure against attacks.

1.1 Motivation

The value of the bitcoin network and the prominence of the new currency has led to the discovery of various attack vectors in the ecosystem. They include the double spending of bitcoins [2] or the separation of a part of the bitcoin network as described in [3]. There were also concerns about the general assumptions of the protocol in [4]. The decentralised infrastructure and the lack of a central authority render protection mechanisms against such attacks difficult to design. Any alteration on the protocol or the clients running it must be incrementally adaptable by the network while keeping backward compatibility.

M.Apostolaki et al. show in [5] that BGP hijack attacks against the bitcoin network are feasible. By corrupting the routing tables of routers, traffic in the internet can be distracted from its original path. These attacks are difficult to detect and hard to protect against, because a single node is not able to tell if it is being attacked or not. On the other hand they are able to affect a large amount of clients simultaneously. An attacker is able to separate the bitcoin network to enable double spending of bitcoins or he can delay the block propagation, which will negatively affect the expected revenue of the miners in the isolated part of the network. As a follow up, the same group proposed SABRE, a secure relay network for protecting the bitcoin network against routing attacks in [6]. They show, that a network of strategically placed relay stations in the internet can offer protection against routing based attacks. Their solution is incrementally adoptable by individual protocol participants.

The proposed infrastructure makes use of relay stations in the bitcoin network. As the future of bitcoin or related cryptocurrencies is not clear, this infrastructure must be able to scale beyond the around 10'000 publicly reachable nodes [1] and the many more nodes that are connecting though a NAT to the bitcoin network. The current state-of-the-art bitcoin relay network FIBRE [7] consists of 5 relay nodes that are geographically distributed across the world and that are connected using fast links. FIBRE reduces the propagation latency by using UDP as a transport protocol and by reducing the need of retransmissions between the relay stations with the help of error correcting codes. However, the fundamental problem of scaling up the relay is not addressed.

1.2 Contribution

This thesis aims at designing the changes needed in the state-of-the-art bitcoin client for it to be able to use the SABRE infrastructure and to scale the relay network up to multiple thousands of peers. It makes the following contributions:

- Profiling of the current standard bitcoin client is performed. We show that the current design is not suited for scaling up. Strong evidence for the bottleneck in the scaling process is presented.
- The bitcoin client is adapted to support the SABRE infrastructure. Additionally, a controller for this infrastructure is created.
- The proposed solution is analysed and we show that it is able to scale with less overhead than the current state-of-the-art bitcoin client.

1.3 Overview

The rest of this thesis is structured as follows: In chapter 2, the most important background knowledge about the bitcoin protocol is presented. In chapter 3, the current state-of-the-art bitcoin client is analysed. We show that the current client has bad scaling properties. We provide strong evidence that the intra-process communication is the limiting factor. The proposed additions to the bitcoin client and the switch controller are presented in chapter 4. In chapter 5 they are evaluated. Furthermore, the scaling behaviour and the costs of the improved scaling capabilities are presented. We conclude this thesis with an outlook in chapter 6 and a summary in chapter 7.

Chapter 2

Background on Bitcoin

In this chapter, the necessary background information on the bitcoin protocol and the implementation of the client is presented. The focus lies on the parts of the protocol and the client that implement the peer-to-peer network.

2.1 Primitives

The fundamental task of the bitcoin protocol is to allow two arbitrary parties to exchange pieces of its currency, the bitcoins. The protocol calls the exchange of bitcoins a transaction.
The bitcoin protocol builds trust in an untrusted environment, it does so, by adding blocks of transactions to a global immutable distributed ledger, called blockchain. This chain is available on nodes participating in the bitcoin network. All transactions that are part of this chain can be verified by every protocol participant and this way, trust in the current state of the chain is generated. The protocol relys on the fact that no single authority controls this blockchain. In other words, no single authority is allowed to append to this chain at will. In bitcoin, this is ensured by the so called "proof-of-work". A mathematically hard problem must be solved in order to be allowed to add a block to the chain. This contains finding a nonce that when included to the input of a hash function, the resulting hash is below a certain threshold. There is no more efficient method to this mathematical riddle as the brute force trial-and-error approach. While finding a nonce which fulfils this requirement is hard, the verification of this proof-of-work is fast. To generate an incentive for protocol participants to try to solve this task, the participant which finds a block receives an amount of bitcoins in form of a mix of a reward and of transaction fees from everyone performing a transaction. This procedure is generally known as "mining". The amount of people and the amount of computing power trying to solve these mathematical quizzes saw a continuous increase in the past 10 years. The protocol handles the automatic adaption of the threshold such that there is an expected delay of 10 minutes between two successive blocks. S.Nakamoto argues in [8] that such a distributed system prevents the double spending of bitcoins, if the majority of the computing work used to find a solution to the mathematical riddle is done by honest nodes.

2.2 Peer-to-Peer protocol

The bitcoin protocol describes a peer-to-peer network which is used to exchange transactions and blocks. Only the most important aspects of the protocol are sketched here. The interested reader is invited to read on in the bitcoin network documentation in [9].
A node in the bitcoin network maintains a given number of incoming (peer initiated) and outgoing (self initiated) connections to other peers. Using these connections, the node is able to receive and distribute transactions and blocks. Transactions are exchanged using a mix of advertisement and request: nodes will regularly send inventory (inv) messages to their peers containing identifying hashes of new transactions. The peer can check, which transactions it does not currently have and requests them from the peer it received the inv message from. Only then, this peer sends the whole transaction.

Using headers messages, the nodes regularly exchange their view on the blockchain. If a node notices, that a peer has a block which itself does not know, it requests the block using a compact block (cmpctblk) message. The peer will then respond with the compact block which contains the block header and a list of transactions that belong to this block. The traditional approach in exchanging blocks was to use the same mechanism as for exchanging transactions. In the second approach, the block is transmitted as a whole while in the first approach, the block is reconstructed at the receiver using the knowledge about the transactions it already possesses. As the legacy way of exchanging messages causes larger message sizes and needs longer to be verified, the compact block is preferred and commonly used. However, the traditional approach is used when the other end of the connection is too far behind the current head of the chain as it is improbable that the peer already has transactions belonging to a specific block.

Ping and pong messages are another message type. To test connectivity and measure the latency to its peers, a node regularly initiates ping messages to which the peer has to respond with pong messages.

The peer-to-peer network protocol allows different other message types which are not important for this work. The documentation for all message types in the current version of the bitcoin client can be found in [9].

2.3 Client structure

To understand the changes made to the bitcoin client, it is important to have a basic understanding of the implementation of the bitcoin client. The focus in this piece of background information focuses on the peer-to-peer network part of version 0.16 of the bitcoin client. The client is multithreaded and the structure that it uses can be used to explain the logical structure of the program. We lay the focus on two threads. The first one is referred to as the SocketHandler thread. This part of the client is responsible handling incoming and outgoing messages. The second one is referred to as MessageHandler thread, which is responsible for processing the incoming messages and to assemble outgoing messages.

We start the explanation with an overview of the message processing pipeline that can be found in the bitcoin client. The explanation will use figure 2.1 as illustration of the path of the messages in the system. The following paragraph includes numbers which link to the individual parts of this illustration. Per connection, the client maintains a data structure called CNode. This CNode is shared between the SocketHandler and the MessageHandler. The client also maintains one socket structure per client. Incoming messages are waiting in the sockets to be read. The SocketHandler regularly checks all sockets for new packets (1). If there are new packets, they are read from the socket and added to a message queue in the shared state between the SocketHandler and the SocketHandler (2). A message can consist of many packets. As soon as one message is finished, a signal is sent to the MessageHandler which reads the message from the CNode (3). The MessageHandler checks the consistency of the message and analyses its payload. According to the message type, different tasks are then performed. For example, transactions and blocks are verified and stored or the local storage is checked for transactions mentioned in advertisements. Finally, the MessageHandler assembles the reply to the peer. If possible, this message is then immediately sent by the MessageHandler (4). If this is not possible, the response is written to the CNode and the SocketHandler is informed that there is a new outgoing message waiting (5). The SocketHandler regularly checks the CNodes for outgoing messages and sends them to the corresponding peer (6).

There is a large amount of shared information between the SocketHandler and the MessageHandler. There is a list of CNodes and there are the CNodes themselves. To prevent access race conditions and keep the state consistent, they are protected using mutexes. Write and read accesses to individual parts of the CNodes require acquiring a lock.

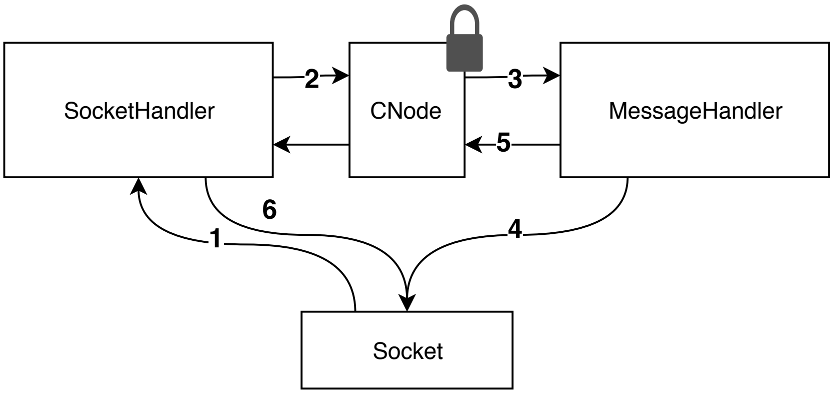


Figure 2.1: Overview over the message handling process in the bitcoin client. This figure illustrates the flow of a message through the networking part of the bitcoin client.

Chapter 3

Profiling

In this chapter, we prove that the current state of the art bitcoin client does not scale well when the number of connected nodes is increased. First, we show general measurement on various system performance indicators. After spotting the CPU to be the possible bottleneck, we perform an in depth analysis of the runtime behaviour of the two most heavy weight threads. This leads to the insight that the inter-thread communication is the bottleneck for scaling the bitcoin client.

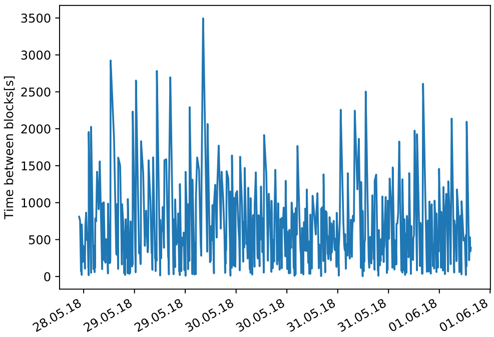


Figure 3.1: Overview of the time between two successive blocks. The time was measured for 650 blocks between 1.6.18 and 28.5.18 and is given in seconds on the y-axis.

3.1 Challenges

Before we present the results, we discuss some challenges in measuring the performance of the bitcoin client. First, the bitcoin network is hard to observe in an isolated setting. The measurements were done with the bitcoin client connected to the bitcoin network. This means, that there are potentially large fluctuations in traffic. For example, as there is no fixed period for new blocks, it sometimes happens that there are no new blocks for 90 minutes. On the other hand, there are times when there are 4 new blocks in 10 minutes. To illustrate this, we plotted the time between two consecutive blocks in figure 3.1. To minimise the impact of such fluctuations, we designed the experiments to be long running and repeated them at least 3 times.
Second, the client does not allow an arbitrary amount of connections. The bitcoin client uses *select* to query the sockets used for the connections to its peer. According to the linux manual page of *select* [10], the total number of sockets that *select* can handle is limited by FD_SETSIZE which is hardcoded to 1024 on Linux distributions. Subtracting the 150 file descriptors that the client reserves for opening files and handling communication, it is possible to open 874 connections to peers. This behaviour could be changed by rewriting the logic of the bitcoin client to use *poll* instead of *select*, but this is not in the scope of this thesis. The consequence of this is that

we were not able to generate an arbitrary load on the client during the experiments. Therefore, finding the bottleneck in the current state-of-the-art bitcoin client is hard.

3.2 Methodology

In this section, the general experimental setup and the tools used are presented. The actual experiments are described together with the results in section 3.3.

Device under test

The experiments were run on two devices:

1. Laptop with Intel dual Core i7 M620 CPU running at 2.67GHz with hyper-threading enabled. 4GB RAM. Running Debian Stretch.
   This device is referred to as *device 1*.

2. Laptop with Intel dual Core i7 6600U CPU running at 2.60GHz with hyper-threading enabled. 20GB RAM. Running Ubuntu Xenial.
   This device is referred to as *device 2*.

Note that *device 2* has a more performant CPU and more memory than *device 1*.

Client under test

All experiments were performed with version 0.16 of the bitcoin client.
Only minor modifications were made to the client. First, the hardcoded connection limits were increased in net.h.

```
        MAX_OUTBOUND_CONNECTIONS = 30
        MAX_ADDNODE_CONNECTIONS = 850
        DEFAULT_MAX_PEER_CONNECTIONS = 1000
```

The values were chosen according to the limitations of the OS and the bitcoin client implementation (see section 3.1).
For some experiments, it is necessary to connect two specific clients to each other. To guarantee that this connections are always established successfully, a minor change was made to always allow incoming connections from whitelisted peers. The following change was made in net.cpp:

```
@@ -1121,7 +1123,7 @@ void CConnman::AcceptConnection(const ListenSocket& hListenSocket) {
             return;
         }

-        if (nInbound >= nMaxInbound)
+        if (nInbound >= nMaxInbound && !whitelisted)
         {
             if (!AttemptToEvictConnection()) {
                 // No connection to evict, disconnect the new connection
```

Using the program *top*, the priority of the bitcoin client process is increased to minimise the effect of background processes on the measurement.

3.2.1 Measurement tools

Various tools were used to measure the resources used by the bitcoin client. For reference, they are listed here.

Ping client

The *ping client* is a modified version of the bitcoin client. This client will ignore most incoming messages and will send a bitcoin *ping* message to the regular client every second. The *ping client* only connects to the regular client and does not maintain or open other connections. The times when a *ping* message is sent and when the *pong* answers arrive are logged. Bitcoin clients by default send a *ping* message every 120s. This time was reduced to one *ping* per

second. Note that the client does not send a *ping*, if there is already a *ping* request pending. Therefore, the actual period is defined as $\min\{1s, \text{RTT}, \text{timemout}\}$. The client does no other work. It does not accept other connections and does not ask for transactions or blocks. This keeps the *ping client* lightweight. The exact changes can be seen in appendix B.
To make sure that the *ping client* is able to connect to the bitcoin client, its ip was added to the whitelisted peers in the configuration of the client under test.

gperftools

gperftools, short for great performance tools or google performance tools, was used to profile the CPU during runtime of the bitcoin client. *gperftools* uses a stack sampling approach for profiling the CPU. A timer fires with a fixed frequency. When the timer fires, the profiler checks which function is currently executed by looking at the current stack frame. This approach leads to measurement inaccuracies, but it allows profiling with low overhead. The profiling time was set to multiple hours and the experiments were repeated multiple times to get statistically significant results. The exact times are given in the detailed experiment setups in section 3.3. *gperftools* was chosen as a profiler because it is well suited for profiling multithreaded applications and produces little overhead. Note that in case of multithreaded profiling, the threads that should be profiled have to call ProfilerRegisterThread() after they are created. This will add a new timer for this thread. Configuration was done using environmental variables. The following settings were used:

```
        export CPUPROFILESIGNAL=12                # signal used to start and stop profiling
        export CPUPROFILE=~/gproftools.prof       # location to save the output profile to
        export CPUPROFILE_FREQUENCY=100           # sampling frequency (samples/s)
        export CPUPROFILE_REALTIME=1              # use the Linux interval timer
        export CPUPROFILE_TIMER_SIGNAL=34         # use another timer signal not to
                                                  # interfer with SIGALARM
```

gperftools creates a profile in a binary format which can be translated to call graphs or that can be imported to kcachegrind/qcachegrind[11]. These tools provide a graphical interface to navigate through the collected data. For more details, the documentation of the CPU profiler of *gperftools* can be found in [12].

vmstat

vmstat is a Linux tool to report the usage of various system resources. *vmstat* reads these information from the /proc filesystem. It is used, because it shows a broad overview of many system parameters such as swap usage, memory usage, disk I/O, interrupt frequency, frequency of context switches, the CPU usage and the I/O wait time. Using *vmstat*, all information is in a common place which makes the analysis easier. More information can be found in the Linux manual page in [13].

top/ps

top and *ps* are standard Linux tool to get realtime information about the current resource usage. They are able to show per thread CPU usage and the total amount of memory used. *top* was used because of its ability to continuously measure. *ps* was used because it the output can be formatted more easily, which helps for the analysis of the experiments. Otherwise, the two programs can be used interchangeably. More information about *top* and *ps* can be found in the Linux manual pages in [14] and [15].

3.3 Experiments

In this section, a series of 4 experiments is presented. First, we measure the performance of a client that is connected to an increasing numer of peers. After experiencing a performance degeneration in this experiment, a second experiment is done which shows the resource usage of the bitcoin client when it is connected to a maximum of 860 peers. The last two experiments analyse the increased CPU usage of the client and show that the inter-thread communication

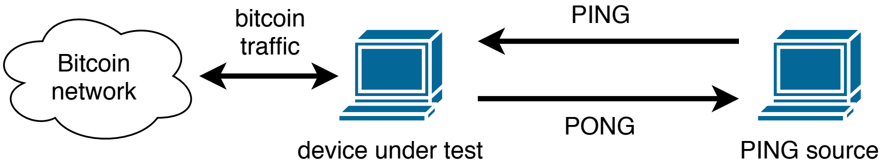


Figure 3.2: Test setup for the performance degeneration test. The *ping client* sends a *ping* every second. The device under test replies with a *pong* message. The times when a *ping* is sent or a *pong* is received are logged at the *ping client*. During the test, the client receives regular bitcoin messages from its peers over the bitcoin network.

is limiting the scaling of the currently used standard client. In the following, the reasoning behind the experiments and the exact setups are explained and the results are presented and explained.

3.3.1 Performance degeneration

The goal of the first experiment is to show that the performance of the bitcoin client degenerates when connecting to many peers. The load on the client is dependent on the current state of the bitcoin network and the peer activity in general (see section 3.1). Furthermore, the load is dependent on the time. For example, a higher load is expected when a block has to be verified. There are 2 network parameters commonly measured to make assumptions about networks. The first of them is throughput. The intention would be that the bitcoin client is not able to process the data fast enough when connected to many peers and would drop packet or would throttle the amount of messages sent. As argued in section 3.1, it was not possible to bring the client to its limits and therefore we could not observe any packet drops or intentional message throttling which makes the use of the throughput as performance indicator difficult to measure and to interpret. The other common performance indicator is the latency. We measured the RTT of a client connected to the bitcoin client under test. The expected result is an increasing delay at the client when it is connected to an increasing number of peers. To measure the performance of the client, the RTT of a *ping*/*pong* exchange was measured using the *ping client*. According to the bitcoin protocol, upon the arrival of a *ping* message, a client must respond with a *pong* message[9].

Setup

The *ping client* runs on the same host as the software under test. The setup is illustrated in figure 3.2. The experiment was conducted as follows:

- The bitcoin client is started.
- The *ping client* is started.
- Iterate over the following desired connection levels: 30, 100, 300, 500, 700, 860 and for each level do the following:
  - Establish new connections until the desired connection level is reached.
  - Wait for 15 minutes to assure that the client is in a steady state.
  - The measurement starts. For 1 hour, the actual number of connected peers is logged and new connections are established, if the connection count drops below the desired level. The logged data is used to match the logs from the *ping client* to a connection count level and to verify that the desired connection count matches the actual number of open connections.

We run the experiment 3 times on both devices. This means that every of the discrete connection levels in the experiment was measured for 3h per device.

Number of connections	RTT *device 1* [μs]	RTT *device 1* [μs]
30	8’198	3’754
100	7’151	5’630
300	26’420	8’930
500	81’797	20’794
700	174’920	30’482
860	187’246	40’246

Table 3.1: RTT measurements of the bitcoin client. The RTT of a *ping*/*pong* exchange with the bitcoin client is measured for different numbers of peer connections.

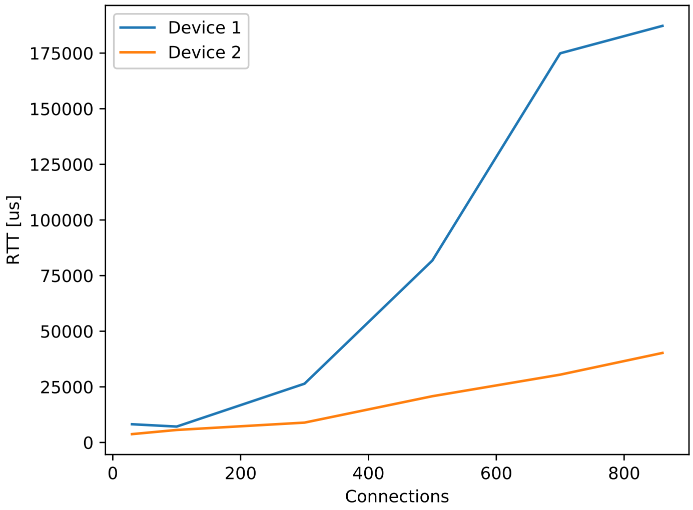


Figure 3.3: This figure shows the measured RTT of a ping/pong message exchange when the client under test is connected to an increasing number of peers. We see that the delay is increasing with the number of connections. After around 300 connections we see a larger increase in the delay for both devices.

Results

The results of this experiment are listed in table 3.1. We are interested in the change of the measured RTT. In figure 3.3, the data is plotted. The plot shows the increase of the delay when additional connections are established at the client. We note that after 300 connections, the additional overhead per added connection gets larger. We assume that the increase is smaller for *device 2*, because the device has better stats. The results show that the RTT is dependant on the number of connections which is a clear sign of bad scaling properties of the bitcoin client.

3.3.2 General Resource Usage

We have seen that the performance of the client becomes worse when more connections are added. To find the limiting factor on the performance, we do another experiment and measure various system parameters. CPU, I/O, Swap and Memory usage were measured.

Setup

- The bitcoin client is started.
- Iterate over the following desired connection levels: 30, 100, 300, 500, 700, 860. For each, do the following:

  - Establish new connections until the desired connection level is reached.
  - Wait for 15 minutes to assure that the client is in a steady state.
  - The measurement starts. For 1 hour, the actual number of connected peers is logged and new connections are established, if the connection count drops below the desired level.*vmstat* is used to measure the CPU usage in user and kernel space, the number of context switches, the I/O wait time, the amount of free memory, the amount of swap used and the amount of kernel buffer space used. One measurement is taken every 5 seconds.

We run the experiment 3 times on both devices. This means that every of the discrete connection levels in the experiment was measured for 3h per device.

Results

An overview of the results can be found in table 3.4. For the CPU measurements we note, that the CPU usage in both devices increases, while the idle time of the CPU decreases. The absolute values for the total CPU usage at 860 connections reported by *vmstat* (mean value over the 3h of experiment) are 52% and 47% for *device 1* and *device 2* respectively. *vmstat* gives the values in respect to the total computing power. However, it is not clear if this load is distributed evenly across the 4 virtual CPU cores both devices have. To check this, the experiment in section 3.3.3 is done.

We observe many context switches. Using *strace*, we see that approximately 90% of the system calls are calls to *futex*. According to Franke et Al. in [16], the call to *futex* is used for fast thread synchronisation.

When looking at the I/O measurements, we note that disk reads and disk writes do not seem to be the bottleneck as the measurements at 860 nodes show a similar amount of disk reads/writes as the measurement at 30 nodes. The same is true for the I/O wait measurement which measures the time a process has to wait for I/O devices in general. We see an increase in interrupts. This is expected as the network interface does receive more messages and will therefore produce more interrupts.

From the data, it is clear that swapping is not causing the performance degeneration. This is also clear as the memory usage peaks at around 35% and there is no need for excessive swapping. Also the buffer space used to store kernel space object does not seem to run out.

With these result, I/O and memory bottlenecks can be ruled out. We noted, that the CPU usage only goes up to around 50%. Next, we will check how the workload is distributed over the cores. This will allow us to pinpoint the bottleneck to the CPU or rule this factor out as well.

3.3.3 Per thread CPU usage measurement

We noticed that the CPU usage is increasing. However, the overall CPU usage only goes up to approximately 50%. As the client is multithreaded, we would expect the load to spread across all 4 virtual cores. However, if the load is not distributed evenly, it could be the case that 1 or 2 cores run at 100%. We do an additional experiment, in order to show that which part of the system is using most CPU time and that the load is not spread evenly across the cores. For this, the CPU usage was measured on a per-thread basis.

Setup

- The bitcoin client is started.
- Iterate over the following desired connection levels: 30, 100, 300, 500, 700, 860. For each do the following:
  - Establish new connections until the desired connection level is reached.
  - Wait for 15 minutes to assure that the client is in a steady state.

  - The measurement starts. For 1 hour, the actual number of connected peers is logged and new connections are established, if the connection count drops below the desired level. *ps* is used with the -L command line option which shows the per thread CPU and memory usage. Measurements are taken every second.

We run the experiment 3 times on *device 1* and *device 2*. This means that every of the discrete connection levels in the experiment was measured for 3h per device.

Results

The results of the experiment are plotted in figure 3.4. The result looks similar for both devices, thus only the results of *device 1* are shown. We see that the SocketHandler thread (bitcoin-net) and MessageHandler thread (bitcoin-msghand) are responsible for most of the CPU usage. At 860 connections, these two threads are using about 90% of CPU time. Again, if the system was loaded, we would expect the CPU time for at least one of these threads to go up to 100%.
The CPU time reported by top is a mean over a certain period of time. We perform a variant of the experiment with additional resolution to get a more fine grained view of the CPU usage of the SocketHandler thread and the MessageHandler thread. For this, we increase the sampling frequency and keep track of how many times the CPU time goes up to 99.9%.

Setup 2

- The bitcoin client is started.
- Connection setup requests are sent to the client using the command line interface. The requests are sent at a fixed frequency of 8 connections per second. During the connection establishment, the CPU usage is measured every 100ms. Additionally, the number of established connections is logged.
- When the client reaches 860 let it run for another minute and then stop.

The experiment was run 3 times on *device 1*.

Results 2

We see in figure 3.5b, that for the SocketHandler thread (bitcoin-net) the number of these "bursts" (moments when the CPU usage is above 99.9%) of CPU usage increases with an increasing number of connections. Interestingly, we observe that starting from about 400 connections the amount of these "bursts" decreases for the MessageHandler thread at about the same rate at which it is increasing for the SocketHandler thread. The reason for this becomes clearer after the next experiment.

3.3.4 Thread work analysis

By now, we have strong evidence that the CPU usage is linked to the bottleneck for the scaling of the bitcoin client on our test devices. The following experiment shows the difference in the workload distribution of the tasks that are performed by the SocketHandler thread and the MessageHandler thread under different load.

Setup

The measurement is done using *gperftools*. The experiment is conducted on *device 1*. The test setup is the following:

- The bitcoin client is started.

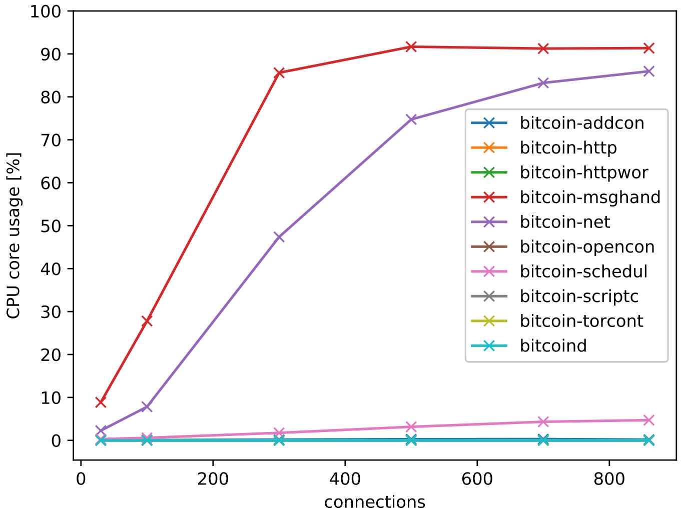
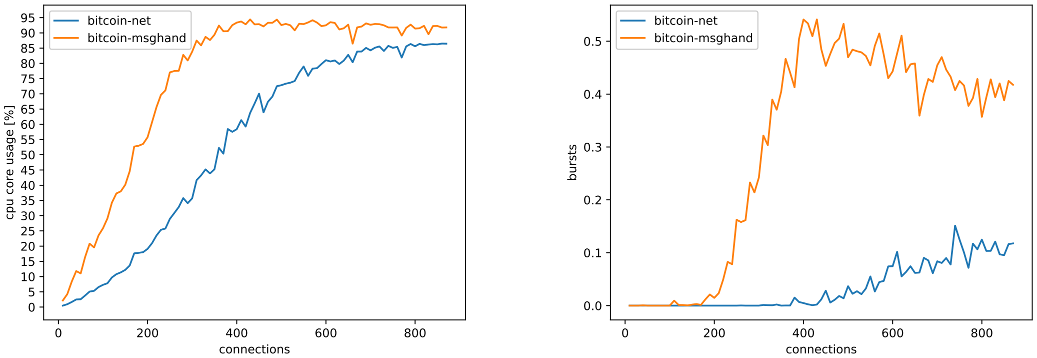


Figure 3.4: CPU usage of the individual threads of the bitcoin client when connected to different numbers of peers. The only two threads using a lot of CPU time are the bitcoin-net and bitcoin-msghand threads. The other threads use less than 5% CPU time.



(a) CPU usage of the MessageHandler and the SocketHandler as a function of the number of connected peers. We see that both threads increase to about 90% CPU time. After that, the CPU time used does not increase anymore when adding additional connections.

(b) Ratio between the measurement points above and below 99.9% CPU usage. By looking at the amount of time the threads are using more than 99.9% CPU time, we get a more fine grained view over the trend of the CPU usage of the threads. At around 400 connections, we see that the CPU time for the bitcoin-msghand thread starts to decrease while the CPU time of the bitcoin-net thread increases.

Figure 3.5: Detailed CPU usage of the MessageHandler and the SocketHandler threads.

- Iterate over the following desired connection levels: 30, 400, 860. For each level do the following:
  - Establish new connections until the desired connection level is reached.
  - Wait 15 minutes to guarantee a normal mode of operation
  - Profile the CPU usage using gperftools for 4 hours. For this, *ProfilerRegisterThread()* was compiled in at the start of the SocketHandler thread and the MessageHandler thread to enable individual timers for these two threads.

This experiment was repeated 3 times. This leads to a total of 12h measurement for 30, 400 and 860 connections, each.

SocketHandler thread

We focus on the three function calls that use the most CPU time when 30 connections are established. Therefore, the results presented will not sum up to 100%. An overview of these functions and what they are used for can be found below.

- ***CriticalBlock::CCriticalBlock*****:** Wrapper that is used for thread synchronisation. Makes the current scope a critical section. A std::recursive_mutex lock is acquired that is automatically released when the scope ends.
- ***__select*****:** In the context of the bitcoin client, *__select* checks for incoming packets and the availability of sending messages by checking the TCP sockets of all connected clients.
- ***CThreadInterrupt::sleep_for*****:** The calling thread waits until the timeout exceeds or the thread is waked by another thread. This roughly translates to "there is nothing to receive or send".

The result of the experiment can be seen in table 3.2. Note that the time spent in *CThreadInterrupt::sleep_for* is a lot smaller at 860 connections than at 30 connections which shows that the thread has to do significantly more work. Intuitively, one would expect the *__select* call to grow with an increasing number of connections. Note that the results are relative numbers and only show the proportions of the function calls. This means, that the actual time spent in the *__select* function can be (and actually is) higher when connected to 860 peers than when connected to 30 peers, because the thread also uses more CPU time in total.
The main observation here is, that when connected to 860 peers, most of the CPU time used by the thread is used to handle the critical sections which are used to communicate with other threads (especially with the MessageHandler thread).

Function	30 conn. [%]	400 conn. [%]	860 conn. [%]
CriticalBlock::CCriticalBlock	1.7	38.0	64.9
__select	16.1	10.7	4.4
CThreadInterrupt::sleep_for	80.8	37.5	3.7

Table 3.2: Difference in CPU time of 3 selected functions of the SocketHandler thread when running with 30, 400 and 860 clients. The unit is the percentage of the cumulated CPU time of the function and all its subfunctions in respect to the total CPU time of the thread.

MessageHandler thread

We focus on the three function calls that use the most CPU time when 30 connections are established. Additionally, we focus on the function *CriticalBlock::CCriticalBlock* because it is using a high amount of CPU time when the client is connected to 860 peers. The results do not sum up to 100% because the some functions are filtered out. Also, *CriticalBlock::CCriticalBlock* is called by various functions in the MessageHandler thread.

- ***PeerLogicValidation::ProcessMessages***: This function handles the message according to the message type. This includes the verification of transactions and blocks, the request of new blocks and transactions and the assembly of responses to the peer. If possible, the messages are sent immediately.

- ***PeerLogicValidation::SendMessages***: Assemble messages that are not a direct response to incoming messages. This includes for example periodic *ping* messages, own transactions or advertisements. If possible, the messages are sent immediately.

- ***std::condition_variable::wait_until***: The calling thread waits until the timeout exceeds or the thread is waked by another thread. This roughly translates to "there are no new messages to process".

- ***CriticalBlock::CCriticalBlock***: Wrapper that is used for thread synchronisation. Makes the current scope a critical section. A *std::recursive_mutex* lock is acquired that is automatically released when the scope ends.

The results of the experiment are summarized in table 3.3. As expected, the functions *PeerLogicValidation::SendMessages* and *PeerLogicValidation::ProcessMessages* increase with the number of connections. This is because more connections lead to more incoming and outgoing messages. From the decrease of *std::condition_variable::wait_until*, we see that the thread spends less time idle. Like with the SocketHandler thread, we observe a clear increase of the overhead of thread synchronising tasks. This increase seems to be lower after 400 connections.

Function	**30 conn. [%]**	**400 conn. [%]**	**860 conn. [%]**
PeerLogicValidation::ProcessMessages	2.6	20.5	31.4
PeerLogicValidation::SendMessages	8.5	66.7	66.3
std::condition_variable::wait_until	88.7	10.4	1.3
CriticalBlock::CCriticalBlock	1.5	30.3	36.5

Table 3.3: Difference in CPU time of 4 selected functions of the MessageHandler thread when running with 30, 400 and 860 clients. Note that the function *CriticalBlock::CCriticalBlock* is not entirely called directly by the main routine of the thread but also include calls from subfunctions of the main routine of the thread. The unit is the percentage of the cumulated CPU time of the function and all its subfunctions in respect to the total computation time of the thread.

Interpretation

From the data we collected we can see the following: First, we see that both heavy weight threads (MessageHandler and SocketHandler thread) spend less time idle while both coming near the limit of 100% CPU core time. Second, we see that both threads use an increasing amount of CPU time for handling the acquiring and freeing of mutex locks. The current implementation of the bitcoin client uses recursive mutexes from the standard library to protect shared resources. Gautham et al. show in [17] that the use of mutexes does not scale when handling many small critical sections. The current implementation of the bitcoin client writes every message into a CNode at least once which will lead to an increasing amount of lock/unlock operations when the number of connections is increased. The next observation is that for the MessageHandler thread, the amount of time spent in CriticalBlock::CCriticalBlock is almost stagnant between 400 connections and 860 connections. At the same time, we observed in the experiment in section 3.3.3 that the CPU usage for the MessageHandler decreases after 400 connections. Together, these two observations lead us to the conclusion that CPU usage is not the bottleneck of the scaling of the bitcoin client, but the inter-thread communication. The increased amount of context switches due to calls to *futex* that we have observed in section 3.3.2 supports this hypothesis.

Category	Parameter	Unit	30	100	300	500	700	860
CPU	User	%	7.35	12.83	34.40	41.34	43.20	43.84
	Sys	%	1.18	1.77	5.24	7.73	8.29	8.55
	Idle	%	90.29	84.82	59.77	50.37	47.86	47.02
	Context switches	/5s	3’012.09	9’542.94	75’166	115’066	121’064	122’702
I/O	Interrupts	/5s	1’117.95	1’492.87	3’538.71	4’913.56	5’702.34	6’187.21
	Disk reads	/5s	409.73	154.80	124.67	157.03	412.13	386.02
	Disk writes	/5s	554.82	208.19	245.07	279.40	309.36	336.65
	I/O wait	s/5s	1.09	0.48	0.38	0.42	0.53	0.51
Swap	Swap	KB	327’516	327’656	327’992	328’202	328’595	327’470
Memory	Free memory	KB	133’188	132’763	259’472	314’197	357’951	540’859
	used Buffer space	KB	108’994	107’321	69’476.4	52’858	39’889.5	45’220.6

(a) Results of *device 1*

Category	Parameter	Unit	30	100	300	500	700	860
CPU	User	%	9.30	14.03	25.74	32.33	34.71	35.18
	Sys	%	2.08	2.82	5.57	9.00	11.22	11.75
	Idle	%	88.24	82.78	68.3	58.33	53.75	52.74
	Context switches	/5s	2’719.2	11’265.9	124’961	234’297	290’314	298’712
I/O	Interrupts	/5s	904.87	1’363.25	5’665.1	9’719.25	12’199.8	12’916.9
	Disk reads	/5s	221.15	1.31	12.75	7.05	3.51	3.80
	Disk writes	/5s	340.51	84.01	129.41	175.03	218.09	245.50
	I/O wait	s/5s	0.39	0.33	0.33	0.33	0.33	0.33
Swap	Swap	KB	256	256	256	256	256	256
Memory	Free memory	KB	1938250	1758470	1355480	890892	324599	204854
	used Buffer space	KB	2’293.33	2’293.33	2’293.33	2’293.33	2’148.7	1’565.18

(b) Results of *device 2*

Table 3.4: Resource usage measurement on both test devices for 6 connection count levels which were each measured for 3 hours.

3.4 Conclusion

It was shown that the current bitcoin client does not scale well when adding additional connections. In this section, the most important drawbacks of the official bitcoin client are listed.

- Thread synchronization: In the experiments that were conducted in sections 3.3.2 and 3.3.4 it was shown that thread synchronization is the largest obstacle for scaling the client to support multiple thousands of connections. Another design with less inter-thread communication should be used to improve scalability.

- FD_SET: The use of select in the client limits the current maximum number of connections to 874. To be able to scale, the client should use another mechanism to handle more connections.

- Per connection state: The current implementation stores a considerable amount of memory in the CNodes. To be able to support large numbers of connections, the per connection state should be reduced to reduce the overall memory consumption.

To resolve these shortcomings of the current bitcoin client will be the design goal of the proposed framework in chapter 4.

Chapter 4

Design

We have seen, that the current bitcoin client does not scale well because of its inter-thread communication. While there are different approaches to speed up or eliminate inter-thread communication, we propose a different solution to the problem. We reduce the tasks of the relay to the bare minimum. The sole purpose of the new client should be to relay blocks. We propose to have a special programmable switch which acts as a block cache. A special firmware is able to translate requests into local lookups. A separate controller hosts the more sophisticated part of the logic, like validating blocks. This controller is responsible for updating the switch with new blocks upon their arrival and informs connected peers about the new block. The design uses UDP as the underlaying transport protocol. The benefit of this is, that due to its stateless nature, a UDP based protocol is able to scale better. As much work as possible is handled in hardware. This framework is called SABRE and is the design of M.Apostolaki and the Networked Systems Group at ETH Zürich [6].
We solve all scaling problems of the current bitcoin client noted in chapter 3: Only one client is effectively connected to the controller at a given time. Therefore, we see from the measurements in chapter 3 that we do not expect a large communication overhead. All other clients are able to request the blocks at line rate from the switch. Because the switch has a clearly defined response to a request, there is no need for inter-thread communication. Therefore there is no scaling problem of the inter-thread communication at the switch as well. The controller is able to handle a theoretical unlimited amount of connections while using only one single socket for the connection to the switch. This solves the limited socket problem that is imposed by the use of FD_SET. As we will show in the evaluation in chapter 5, the design also uses considerably less memory per connection.
This thesis focuses on the modifications that are necessary on the clients and the controller. In this chapter, the message protocol is presented in 4.2. The design of the client and the controller is presented in sections 4.3. The chapter is concluded with a discussion of the shortcomings of the current design in section 4.4.

4.1 Terminology

The design in this chapter and the following evaluation in chapter 5 use 3 different bitcoin clients. To avoid misconceptions, they are briefly explained here. For a more throughout explanation of the modified clients, see sections 4.3.2 and 4.3.3. Also, a few other commonly used terms are defined here.

Regular Bitcoin Client

The term *bitcoin client* is used for the unmodified bitcoin client from the official repository on GitHub [18]. Version 0.16 (commit bf3353de90598f08a68d966c50b57ceaeb5b5d96) was used throughout this work. We use the term *regular* when referring to the unmodified bitcoin client or the unmodified bitcoin protocol.

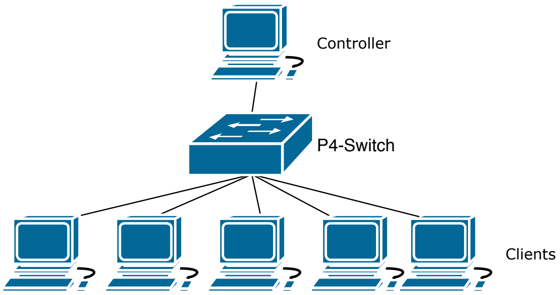


Figure 4.1: Overview of the different participants of the relay protocol.

Modified Bitcoin Client

The *modified bitcoin client* is based on the *bitcoin client* and is augmented with additional features. This allows the modified bitcoin client to connect to the switch relay. The modified bitcoin client can be used as a drop-in replacement for the bitcoin client.

Controller

The *controller* is based on the *bitcoin client* and is augmented with additional features. It is used to interface with the p4 switch and to modify its state. This code is not intended to run on a regular bitcoin node.

Network protocol

We refer to the regular bitcoin peer-to-peer network as the *network protocol*.

Relay protocol

We name the here presented addition to the communication logic *relay protocol*, as it differs completely from the *network protocol*.

4.2 Protocol overview

The protocol that is presented here, differentiates among 3 parties. An overview can be found in figure 4.1. The first group consists of the clients. We assume there to be many clients. The clients have 2 goals. They want to receive new blocks and they want to redistribute blocks that they have learned about. The second group consists of the switch. The switch is a piece of programmable hardware. It acts as a cache. Answers to a predefined set of requests can be delivered at line rate. The last protocol participant is the *controller*. The *controller* is used to handle uncacheable requests and to do the more sophisticated computation. It is also responsible for writing the data to the switch.
The protocol that connects these 3 groups is now presented. The design of the system and the network was done by M.Apostolaki and the Networked Systems group at ETH Zürich.
The protocol can roughly be split into 3 parts which are illustrated in figure 4.2. Unless otherwise stated, UDP datagrams are used as underlaying transport protocol. The first part of the protocol (see figure 4.2a) handles the connection setup between the client and the switch. A 3-way handshake is used to establish the connection. The client initiates a handshake. After receiving the handshake request, the switch replies with a message containing a secret value. To complete the handshake, the client sends another message containing the same secret back to the switch. The secret is used to protect the switch from spoofing. After the handshake is complete, a message containing the connection details of the client is sent to the controller to store this metadata.
The second part of the protocol (figure 4.2b) handles the update process of the switch. When a *modified bitcoin client*, which is connected to the switch, receives a new block, it advertises

it to the switch. If the switch does not know this block, it will tell the client to connect to the controller at a specified port/IP. This connection is established using the regular bitcoin protocol. After the client has connected to the controller, it will send the new block to the controller. After that, the connection is closed and the controller updates the switch. It will first send a message indicating, that a new block will be sent. After that, the block is sent in chunks of 499 bytes to the switch. These block fragments are called segments. The last segment is padded to have the same size. This is needed because of limitations of the switch. A precomputed UDP checksum is added, to reduce the computation that the switch has to do on an incoming request.
After the controller updated the switch, it will advertise the new block to all connected clients. Part 3 (figure 4.2c) of the protocol explains how blocks are distributed to the clients. The controller sends an advertisement containing the hash of the block and the total number of segments, the block was split into. The clients that are interested in the block can request the individual segments from the switch directly. From this point on, the controller does not have to do any more work for distributing this block. A client has to request all segments seperately to be able to reconstruct the block.

4.2.1 Message types

The *relay protocol* uses a set of message types for the communication between client, switch and controller. The message types and their use are listed here for reference. To see, in which part of the protocol the messages are used, see figure 4.2 The focus lays on the functional aspects of the message types. The exact message structures and sizes are listed in appendix A. For better readability, the message types of the *relay protocol* are written using capital letters throughout this thesis.

- REY: This message type is used to establish a connection between the client and the switch. A 3-way handshake is performed. The messages of the individual steps of this handshake are marked using a flag.

- INV: The INV message is used by the controller to advertise new blocks. It contains the block hash and the number of segments the block is split into.

- SEG: Using the SEG message, the client requests a segment of a block from the switch. For this, the message contains the hash of the block and the segment number that is requested.

- BLK: The BLK message is used by the controller to update the switch. The message contains the segment ID and the data. Additionally, the packet contains a precomputed checksum which is used by the switch to create the UDP checksum when the client requests the block. Note that the UDP checksum depends on the destination port and can therefore not be computed in advance and must be calculated by the switch.
  The switch sends BLK messages as reply to SEG messages from the client. These BLK messages only contain the segment ID and the data.

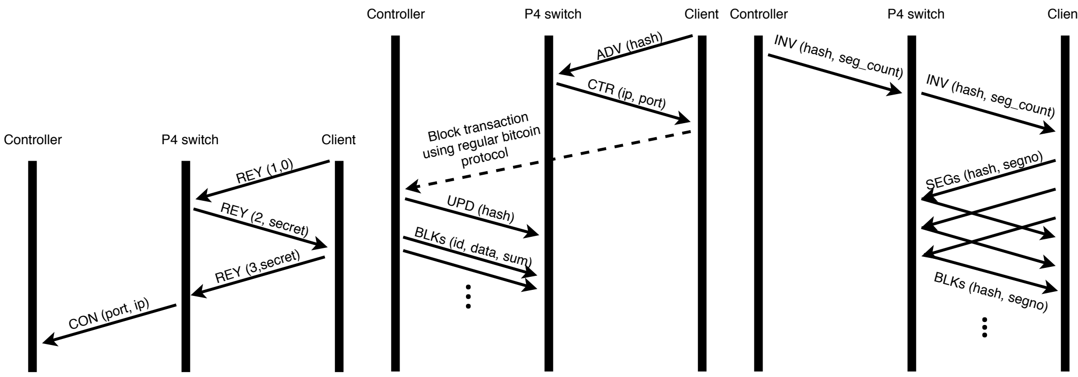
- ADV: The client sends an ADV message to the switch to indicate that it received a new block. This message contains the hash of the new block.

- CTR: The switch may respond to an ADV message with a CTR message, indicating that the client should connect to the controller. The message contains the IP and the port which should be used to connect to the controller.

- UPD: The UPD message is sent by the controller. The switch uses this message as an indicator, that the controller wants to add a new block. The message contains the hash of the new block.

- BCL: The BCL message is used to blacklist a peer and is sent from the client to the switch.

- CON: After a successful handshake with a new client, the switch sends a CON message with IP and port of the client to the controller. This message is used to inform the controller about new connections.



(a) Handshake (b) switch update (c) client side block request

Figure 4.2: Overview of the individual phases of the *relay protocol*.

4.3 Client and Controller Implementation

The changes to the bitcoin client aim at being lightweight and modular. First, we describe some general design choices that are common to both the *modified client* and the *controller*. Next, we discuss the peculiarities of the *modified client* and the *controller*.

4.3.1 General design

Both, the *modified client* and the *controller* are based on the *regular bitcoin client*. The benefit of this is, that parts of the existing code can be reused. One decision in the design process was to have most of the logic inside a separate C++ class to be able to quickly get an overview of the additional logic.

For better testability of the design, the 3 software types, namely the *regular client*, the *regular bitcoin client* and the *controller* can be chosen at startup without the need to recompile. This results in a minimally larger binary compared to an approach where the type has to be specified at compile time. Besides the client type, other new command line or config file parameters were introduced. They allow to set the IP address and the port of the switch, as well as the own port that is used to connect to the relay.

Thanks to the socket abstraction, the logic for incoming messages can be reused. After a packet is read from the socket, the code checks if it is coming from the switch. If this is the case, it is handed over to the relay network message handler instead of the normal message handler. For sending, the handler checks if the outgoing message should use UDP before sending because the logic for sending is slightly different between TCP and UDP. Since UDP is connection-less the sender has to provide the destination IP and port.

4.3.2 Modified Bitcoin Client

The modified client runs on a users device. Along the regular tasks of a regular bitcoin client, the modified client also fulfills the here mentioned tasks.

Connection setup

When the modified client starts its networking routine, a Net_relay object is generated. This object is handles the logic that is used to interface with the switch. A new UDP port is created with the connection settings found in the settings. Using the command line or the config file, the default settings for ip address and port of the switch can be overwritten. In a productive setting, anycast could be used to find a relay switch.

The CNode construct of the unmodified bitcoin client was used to enhance the integration with

```
while(not all segments received):
        if(time since last segment > timeout threshold):
                request retransmission of all non yet arrived segments
        else:
                request retransmission of all blocks with id < latest segment - 10
        wait 1s
```

Figure 4.3: Pseudocode for the segment timeout handler. The handler uses a sliding window to trigger the retransmission of lost segments.

the existing codebase. This allows, for example, to get a report over the connected switch using bitcoin-cli as it is possible with regular connections. To be able to differentiate between the UDP connection used for the switch and the TCP connections of regular peers, a isRelay flag was added to the CNode class.

After the CNode is created, the client tries to establish a connection to the switch. As long as the client is not connected to the switch, every minute, a handshake is initiated by sending a REY message. The switch will answer with another REY message containing a secret. Using the same secret, the 3-way handshake is concluded. After the handshake, the relay connection will go to a connected state which will activate all other switch specific functionalities.

The fact that the same data structure could be used for the connection to the relay and for peer connections, means that the relay connection does not introduce more memory overhead than a regular peer connection. Furthermore, because the relay connection uses UDP instead of TCP, less state has to be maintained in kernel space.

Incoming message handling

The bitcoin client checks for incoming messages using FD_SETs. Because of the socket abstraction, the same mechanism as for checking the other connections can be used. The distinction begins when the data is received. A separate handler is responsible for the arrival of messages from the switch. The new code was not included in the main message handling routine, because the message format is completely different and this way, the code for both connection types is logically separated.

INV messages: An incoming INV message is first translated into an inv message from the *regular bitcoin protocol*. This allows us to reuse the checks that are done on regular messages. These checks tell, if the client is interested in this block or will ignore the message.

If the client does want the block, it is requested from the switch. A small data structure is created which holds the hash and the number of segments and the segments themselves. Two timestamps are added which are used to check the soft and hard timeout of the segments. Using these timestamps, a timeout handler for the block is implemented as can be seen in figure 4.3. All segments are then requested from the switch. A small delay of 1ms between the SEG messages is introduced, which leads to a drastically lower packet loss rate because of less congestion at the switch. Because the switch is emulated in software, it is not able to process the requests at line rate and starts to drop packets if they arrive in too fast succession.

BLK messages: When a BLK message arrives, it is checked if the containing segment belongs to a block that the client requested from the switch. If it does, it is added to the list of arrived segments. As soon as the block is completed, it is given to the MessageHandler thread to be processed. From this point, the logic is the same as if the block was received over the regular bitcoin network.

CTR messages: An incoming CTR message means, that a connection to the controller should be established. The already existing mechanism for connecting to a peer can be reused. A flag indicating that the new connection is a "manual connection" is added to the function call to make sure that the client connects to the controller.

The other message types are ignored on the client and are silently dumped.

4.3.3 Switch Controller

The controller is a *regular bitcoin client* which has the extra task to manage updates of the switch and to advertise new blocks. The decision to design the controller as an addition to the

```
udpPreChecksum(bytes):
    // Calculate the sum
    sum = 0;
    len = len(bytes)

    // sum up the 16 bit values
    for(i=0; i<len/2; len+=2):
        sum += bytes[i]<<8 + bytes[i+1];
        if (sum & 0x80000000):
            sum = (sum & 0xFFFF) + (sum >> 16);

    // Add padding if the packet length is odd
    if ( len & 1 ):
        sum += bytes[len-1];

    // Add the pseudo-header
    sum += IPPROTO_UDP;
    sum += length + 8;
    sum += length + 8;

    // Add the carries
    while (sum >> 16):
        sum = (sum & 0xFFFF) + (sum >> 16);

    return sum;
```

Figure 4.4: Pseudocode of the precalculated UDP checksum.

regular bitcoin client is based on the fact that the block validation logic can be reused. This way we make sure that the switch is updated with legitimate blocks and is not updated with arbitrary data. Otherwise, the setup could be used by a malicious user as free anonymous cloud disk space which he could fill with malicious or illegal data.
The controller receives CON messages, when a new client successfully connects to the switch. A CON message contains the IP address and the port of the client. This metadata needs to be stored. The std::set data structure was chosen, as it offers a complexity of $O(\log(n))$ for lookup and insert.
If a new client connects to the controller, client and controller use the regular bitcoin protocol to exchange blocks and transactions. The goal of this connection is for the controller to learn a new block. When the controller received the block from the client, the connection is closed by the controller. The connection is also closed after a timeout of 120s.
After a new block is received that can be attached to the longest chain, the switch has to be updated. For this, the controller sends an UPD message, followed by the block. According to the messaging protocol, the block has to be sent in segment with 499 bytes of data each. The last segment is padded to have a 499 byte length as well. The switch needs a precomputed UDP checksum to be able to send BLK responses to SEG requests by the client. At this point, the final checksum cannot be calculated, as the port of the client is part of the checksum. Figure 4.4 shows the pseudo code for creating the precalculated UDP checksum. The protocol does not specify an answer to the update process of the switch. This means that the controller does not get any feedback if the update was successful or not. During the evaluation of the setup using mininet, we observed that the switch drops packets if they arrive in fast successive bursts. Therefore, the current controller will immediately send a fragment as soon as it is ready and not first queue it and let it send by the SocketHandler thread. Additionally, while sending, a 1ms delay was added between two successive segments. It is not clear, if the delay is needed or would have to be adapted in a real world scenario with actual hardware.
After the switch is updated, the controller advertises the block to all clients for which it has an entry in the connection metadata set.

4.4 Design shortcomings

In this section, some shortcomings of the design are discussed. Part of them is due to limitations of the *relay protocol* (see section 4.2.1) and some are due to the fact that the current design is a prototype.

Missing error handling

If for some reason, a packet is dropped during the update of a switch, the update is not successful. However, the controller does currently not know that the update was not successful. It sends out INV messages but the clients receiving the INV messages will not be able to correctly reconstruct the block from the segments. To tackle this, the switch should implement some feedback on the update process such as the number of segments received and it should check if it receives all segment ids.

Increasing state

The current implementation of the *controller* does not specify a timeout after which the metadata about the connected clients should be deleted. This means that there is a protocol based attack vector. It is possible to fill up the free memory of the controller by performing handshakes with the switch using a large amount of possible IP addresses and port pairs. While the switch offers DOS protection, the attack is still successful if it is performed over a large timespan. To protect against this, the controller should implement a maximum lifetime of the client entries that it stores. Another possibility would be to create a list of fixed length and to remove the oldest entries if a new client connects. Currently, the information about the clients is never removed.

Multiple block update

Currently, only the update of a single block is supported. This means, that when a client which has multiple new blocks connects to the *controller*, only the newest one is sent to the switch. This is of course not practical for a real world deployment. This behaviour can easily be changed as soon as the switch supports multiple blocks.

Windows and IPv6 support

The prototype was tested on Linux and on MacOS. It was not tested if it runs on windows hosts. The current *relay protocol* does not have support for IPv6 addresses which are therefore not supported by the client.

Chapter 5

Evaluation

The implementation of the modified client and the controller was tested using mininet. The evaluation focuses on the modified client and the controller which are the results of this thesis. The software for the switch was written by M.Apostolaki from ETH Zürich.
The evaluation starts with a general setup in 5.1. This shows that the protocol is working as expected. Additionally, we measure the speed of the block distribution. After analysing the scaling properties of the protocol in section 5.2 and comparing the message sizes of the proposed solution to the state-of-the-art bitcoin client in 5.3, we measure the memory and CPU overhead in sections 5.4 and 5.5 respectively.

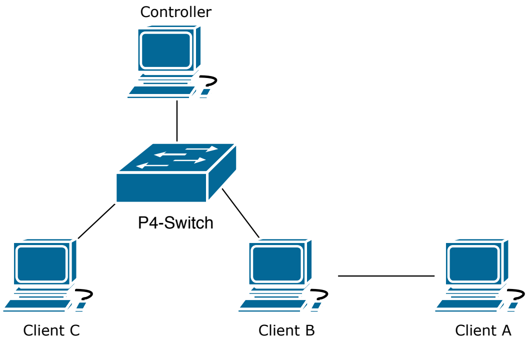


Figure 5.1: Overview of the test setup used for the evaluation. Client A is a *regular bitcoin client* that has 1 block that clients B/C and the*controller* do not posses. Clients B and C are *modified bitcoin clients*. The switch is a programmable P4-switch emulated in software.

5.1 Functional evaluation

Using mininet, a small test setup was created. The setup consists of 3 clients, one switch and a controller. An overview of the setup can be seen in figure 5.1. Client A is a *regular bitcoin client* and clients B and C are *modified bitcoin clients*. The controller is a bitcoin relay *controller* as described in section 4.3.3. The test covers 2 important use cases. First, it shows that the *modified client* is able to interact with the *regular bitcoin client*. Second, the update and advertisement process described in the *relay protocol* is tested.

Setup

The experiment goes through the following steps.

1. Client A has a block that the other clients and the controller do not yet have in their local storage.

2. Clients B and C do a handshake with the switch.

3. Client A establishes a connection to client B.
4. Client A advertises its latest block to client B using the regular bitcoin network protocol.
5. Upon receiving the block, client B advertises the block to the switch.
6. The switch does not know the block and sends a CTR message to client B indicating that it should connect to the controller.
7. Client B establishes a connection to the controller.
8. Client B sends its latest block to the controller using the regular bitcoin network protocol.
9. The controller updates the switch and advertises the new block to all clients.
10. Client C does not have the block in its local storage and requests the block from the switch and adds it to its chain.

Note that there is no direct connection between clients B and C. To make sure that clients A and B establish a connection, the IP of client A was whitelisted at client B.

This setting was run with 10 different blocks and was repeated 20 times for each block which results in a total of 200 experiment runs. The mean block size was 1128194 bytes during the experiment. This means that there are 2261 segments in average.

Results

After each run, the log files that are generated by the clients and the controller are analysed. From this data, we can calculate the time needed for the block to travel from node B through the controller and the switch to client C.

We are interested in the time client B needs to connect to the controller. For this, we measure the time between the moment when client B adds a new block to the chain until the moment when the connection between the client and the controller is established. We see that after 2ms, the client sends an ADV message and receives a CON response from the switch 2ms later. The client then needs 11ms to connect to the controller. So in total, the client is connected 15ms after it received the block. After connecting, the client and the controller need 4.7s to transfer the new block from the client to the controller.

At the controller, we want to measure the time needed to update the switch. For this we measure the time starting from the moment when the controller adds the new block to its chain until the last BLK message is sent. We focus on the work done at the controller and neglect the time of flight of the packet and the overhead at the switch. We measure an update time of 3.4s. Remember that a 1ms delay was introduced when sending the block segments. With 2261 segments this means that there is a 2.26s delay which explains a part of the overhead. 7ms after the last segment is sent, the RINV message to the client is sent.

Lastly, we want to see how fast client C can request the block from the switch. For this, we measure the time between the moment when client C receives the RINV message and the moment when the block is added to the chain. We see, that the last segment of the block arrives after 4.3s. Again, the 2.26 seconds of transmission delay has to be taken into account here. After that, 3.9s is needed to verify the block and add it to the chain. The overhead here stems from the fact, that the client has to verify all transactions that are contained in the block because they were not previously available to the client. In total, the block request takes 8.3s from the moment the RINV message is received. This is 52% slower than the transmission of the block from client B to the controller that was using the regular bitcoin protocol. The whole process, starting from the moment client B receives the block until client C receives it takes 16.3s. To speed up this process, one could first investigate, if the 1ms delay is needed in actual hardware. This would reduce the measured time by 4.5s. Further, a future adaption could allow storing of transactions at the switch which would reduce the time needed by client C to add the new block to its chain. However, most of the time overhead can be regarded as setup cost for the cache at the switch. A more realistic scenario would be, when there is not a single client C but many clients that are interested in the block and are able to request it in parallel.

category	Parameter	Unit	30
CPU	User	%	1.36
	Sys	%	0.79
	Idle	%	97.43
	Context switches	/5s	1'454.08
I/O	Interrupts	/5s	673.37
	Disk reads	/5s	150.35
	Disk writes	/5s	26.87
	I/O wait	s/5s	0.48
Swap	Swap	KB	7.21
Memory	Free memory	KB	114'138
	used Buffer space	KB	67'954.4

Table 5.1: General resource usage measurement on *device 1* when connected to 10'000 nodes via the relay.

5.2 Scaling

To inspect the scaling behaviour of the controller, a python based pseudo client is crated which will open 1'000 connections to the relay. The test setup is the same as in section 5.1, but with client C replaced by 1'000 clients. After the pseudo client has set up the connections, it waits for incoming RINV messages and logs the arrival time of the last incoming RINV message. We run the test with 3 different blocks and 3 runs each. The time needed to send the 1'000 RINV messages was 1.24s. There is a 1ms delay between each message which accounts for 1s of the time needed. If the delay can be removed when working with real hardware, the 1'000 RINV messages are expected to be sent out in 240ms. As we expect the switch to be able to process requests at line rate, we do not expect a large overhead for the clients to request the blocks from the switch. However, this claim could not be verified with the software emulated switch.
Next, the connection count is further increased and the system stats are measured. We use the same setup as in sections 3.3.1 and 3.3.2 but use the relay controller. The measurement is made on *device 1*. We let the setup connect to 30 regular nodes and use a python client to add 10'000 relay connections by sending emulated CON messages. The rest of the test setup is equal to the one defined in the before mentioned sections. CPU, I/O, Swap and Memory measurements show no significant overhead over the setup without the 10'000 relay connections. The results of the measurement can be seen in table 5.1. The measurement of the RTT shows 19'335μs, which is comparable to about 200-300 regular connections according to table 3.1.

5.3 Bandwidth overhead

The current bitcoin protocol specifies different ways to transmit blocks from one client to another. Most of the optimisations aim at smaller block messages to be able to speed up the block distribution. The optimisations relay on prefetching transactions so that the client is able to pre-verify the transactions. This makes it difficult to compare the message of the relay transmitted block size to the one needed by the current bitcoin protocol. Instead, we calculate the overhead compared to a traditional block message.

Client

First, we focus on the overhead introduced by the protocol. The relay message protocol specifies a payload size of 499 bytes. If we define s_i as the size of the block message (block + header) and s_r as the total cumulated message size of all segments, we can express s_r by adding the overhead of the new header per segment and the padding overhead to s_i

$$s_r = s_i + \left\lceil \frac{s_i}{499B} \right\rceil \cdot 5B + s_i \text{ mod } 499 \tag{5.1}$$

Where $\%$ is used for the remainder of the integer division. Currently, blocks have about the size of 1MB. If we assume $504B \ll s_i$, we can express the protocol overhead as

$$\frac{s_r}{s_i} \approx 1 + \frac{5}{499} = 1.01 \tag{5.2}$$

The 499 maximum payload size is smaller than the most commonly used MTU of TCP. This means, that the transport layer also adds an overhead to the transmitted block size. We assume a commonly used MTU of 1500 bytes for the calculations. Further we assume 24 bytes Ethernet headers, 20 bytes IP headers, 20 bytes TCP headers and 8 bytes UDP headers. We denote the transmitted message size of the regular and the relay blocks as t_i and t_r respectively. We get the following expressions

$$t_i = s_i + \left\lceil \frac{s_i}{1500} \right\rceil (24 + 20 + 20)B \tag{5.3}$$

$$t_r = s_r + \left\lceil \frac{s_r}{499} \right\rceil (24 + 20 + 8)B \tag{5.4}$$

Assuming $1500B \ll s_i$ and $499B \ll s_r$, we can estimate the bandwidth overhead as

$$\frac{t_r}{t_i} \approx \frac{s_r}{s_i} \cdot \frac{1 + \frac{52}{499}}{1 + \frac{64}{1500}} = 1.07 \tag{5.5}$$

This rough estimate shows that we can expect overhead of about 7% when comparing to a minimal TCP based transmission. The overhead could be reduced if the protocol would be changed to support larger payloads than 499 bytes. However, as the discovery of a new block is a sporadic event, this should not pose a large drawback.

So far, only the bandwidth overhead of incoming messages were observed. As each segment of the block has to be requested, the protocol also generates overhead on the outgoing messages. We compare the total amount of outgoing messages to a single getdata message of the regular bitcoin protocol. Getdata was chosen over other block exchange mechanisms such as headers messages, because getdata has the smallest foodprint. This will give us an estimate on the upper bound on the overhead. The getdata message with a single block hash has a size of 61 bytes. Per 499 bytes of block, the modified client has to send a request of size 37 bytes. On the other hand, the regular client sends TCP ACKs. If we assume a block of size 1MB, a TCP MTU of 1500B, 24B ethernet headers, 20B IP headers and 8B UDP headers and assume a lossless transmission, we get the following overhead:

$$\frac{\left\lceil \frac{1MB}{499B} \right\rceil \cdot (37B + 52B)}{61B + 64B + \left\lceil \frac{1MB}{1500B} \right\rceil \cdot 64B} = 4.2 \tag{5.6}$$

In other words, the request size is increased by the factor 4.2 when comparing the relay protocol to the regular bitcoin protocol. This is limited by the protocol and can only be changed when changing the protocol or when increasing the data chunk size of the segments. However, as the requests of the individual segments is pipelined, this does not lead to a large overhead time wise.

Controller

To send blocks, the controller has to update the switch. Similarly to equations 5.2 and 5.8, the overhead can be calculated. Here, we have to add the overhead from the UPD message (35 bytes) and the additional checksum in the headers of the BLK messages (+2 bytes). However, as we have $35B+ \ll s_i$ and $\frac{5}{499} \approx \frac{7}{499}$, we get

$$\frac{s_r}{s_i} \approx 1 + \frac{7}{499} = 1.014 \tag{5.7}$$

and

$$\frac{t_r}{t_i} \approx \frac{s_r}{s_i} \cdot \frac{1 + \frac{52}{499}}{1 + \frac{64}{1500}} = 1.074 \tag{5.8}$$

for the outgoing message overhead.

The update process has to be done once per block. Except from the single 37 byte INV message per client, there is no additional overhead when the block is sent to multiple clients. This is a clear benefit when compared to the regular client, which needs to send the block for each peer that requests it.

5.4 Memory overhead

Let's first inspect the memory usage of the controller. Let m_0 be the amount of memory needed to store the state of a single peer-to-peer connection. Let m_b^n and m_r^n be the amount of memory needed to store the state of n peer-to-peer connections and n relay connections, respectively. To maintain the basic connection to the relay, we use the same data structure as used by peer-to-peer connections. The connection uses a UDP connection instead of a TCP connection which uses less state, therefore we can say that

$$m_r^0 \leq m_b^1 = m_0 \tag{5.9}$$

When adding new connections, a new data structure is added for the regular connections. For the relay connections, the metadata of the the peer is stored in the existing data structure. This leads to the following overhead

$$m_b^n = n \cdot m_0 \tag{5.10}$$

$$m_r^n = m_0 + n \cdot 6B \tag{5.11}$$

While the actual size of the data structure can vary across compilers and operating systems, with gcc 5.4.0 on Ubuntu 16.04.1 we have $m_0 = 1672B$. This means that we use less than 1% of the size of regular connections for the relay connections.

On the client, the connection to the switch uses the same data structure as regular connections. This means, that the memory usage is the same in user space. In kernel space, the memory overhead is smaller, because the UDP connection that is used needs to store less state.

5.5 CPU overhead

To measure the CPU overhead of the design, we use the same test setup as in section 3.3.4. We let the node establish 30 connections and add 10000 relay connections using a pseudo client which emulates the CON messages from the switch before starting the profiler. From the results we can see that there is no significant overhead in adding 10000 relay connections. The addition to the system accounts for 3.1% of the total CPU time of the MessageHandler thread.

5.6 Conclusion

With this evaluation, we show that the presented solution has better scalability characteristics than the state-of-the-art bitcoin client. It was shown that the controller can handle multiple thousands of connections with a minimal memory and CPU overhead. The tradeoff for this increased scalability is the increased bandwidth needed by the clients and the slower block propagation.

Chapter 6

Future Work

The proposed modifications to the bitcoin client and the design of the controller can be used in the SABRE framework to protect against routing based network isolation attacks. The next step would be to perform a real world experiment using physical hardware for the switch. If this experiment shows promising results, the presented prototype can be used as a starting point for a real world implementation by addressing the design shortcomings mentioned in section 4.4.

It would be interesting to see if modifications used by FIBRE can be used to improve the SABRE framework in terms of update and distribution speed. Additionally, the update delay of the SABRE framework could be improved by adding support for transactions and compact blocks in a future iteration.

Further investigation could be done on the question, if a switch based caching mechanism could also make sense in other areas, for example as network cache in data centers.

Chapter 7

Summary

In this thesis we proposed a modified bitcoin client for the use with the SABRE framework. In chapter 3, we have shown that the current state-of-the-art has bad scaling properties. Using various measurements we were able to gradually pinpoint the bottleneck in scaling up to be the inter-thread communication. We defined properties that a modified bitcoin client should have. We presented a design for a modified bitcoin client that has these properties in in chapter 4. Additionally, a controller for the switch used in the SABRE network was designed. The shortcomings of the current design and security concerns were briefly discussed in section 4.4. Using our prototype, we were able to show the improved scaling properties in chapter 5 by testing our implementation with up to 10'000 connections. We also showed, that the controller has desirable scaling properties in terms of memory usage and CPU overhead and showed the tradeoff between the scalability and the block distribution speed. We concluded this thesis with a short discussion about possible research directions in chapter 6.

Appendix A

Relay Messaging Protocol

The following table shows the exact specification of the different message types used in the *relay protocol*. The field sizes and the communication directions are given in the corresponding columns.

Message Type	Communication	Size (bits)	Field
REY	C ↔ S	24	Command (=REY)
		4	Flag
		20	Secret
INV	M → C	24	Command (=INV)
		256	Block hash
		16	Segment count
SEG	C → S	24	Command (=SEG)
		256	Block hash
		16	Segment id
BLK	S → C	24	Command (=BLK)
		16	Segment id
		499*8	Segment data
BLK	M → S	24	Command (=BLK)
		16	Segment id
		499*8	Segment data
		16	precomputed UDP checksum
ADV	C → S	24	Command (=ADV)
		256	Block hash
CTR	S → C	24	Command (=CTR)
		32	IP
		16	port
		8	empty
CON	S → M	24	Command (=CON)
		8	empty
		16	port
		32	IP
UPD	M → S	24	Command (=UPD)
		256	Block hash
BCL	M → S	24	Command (=BCL)
		32	ip

M: Controller
S: Switch
C: Client

Appendix B

Ping Client Modifications

The following code alterations are made to create the *ping client* from the regular bitcoin client.

```
diff --git a/src/net.h b/src/net.h
index 8378a303b..cd1dda236 100644
--- a/src/net.h
+++ b/src/net.h
@@ -38,7 +38,7 @@ class CScheduler;
 class CNode;

 /** Time between pings automatically sent out for latency probing and keepalive (in seconds). */
-static const int PING_INTERVAL = 2 * 60;
+static const int PING_INTERVAL = 1;
 /** Time after which to disconnect, after waiting for a ping response (or inactivity). */
 static const int TIMEOUT_INTERVAL = 20 * 60;
 /** Run the feeler connection loop once every 2 minutes or 120 seconds. **/
diff --git a/src/net_processing.cpp b/src/net_processing.cpp
index bf9307727..de2ced1ae 100644
--- a/src/net_processing.cpp
+++ b/src/net_processing.cpp
@@ -2720,6 +2720,7 @@ bool static ProcessMessage(CNode* pfrom, const std::string& strCommand, CDataStr

     else if (strCommand == NetMsgType::PONG)
     {
+        LogPrintf("THROUGHPUT: PONG: %u\n", GetTimeMicros());
         int64_t pingUsecEnd = nTimeReceived;
         uint64_t nonce = 0;
         size_t nAvail = vRecv.in_avail();
@@ -3175,6 +3176,7 @@ bool PeerLogicValidation::SendMessages(CNode* pto, std::atomic<bool>& interruptM
         }
         if (pto->nPingNonceSent == 0 && pto->nPingUsecStart + PING_INTERVAL * 1000000 < GetTimeMicros()) {
             // Ping automatically sent as a latency probe & keepalive.
+            LogPrintf("THROUGHPUT: PING: %u\n", GetTimeMicros());
             pingSend = true;
         }
         if (pingSend) {
@@ -3194,6 +3196,8 @@ bool PeerLogicValidation::SendMessages(CNode* pto, std::atomic<bool>& interruptM
             }
         }

+        return true;
+
         TRY_LOCK(cs_main, lockMain); // Acquire cs_main for IsInitialBlockDownload() and CNodeState()
         if (!lockMain)
             return true;
```

Bibliography

[1] "blockchain.com." https://www.blockchain.com, May 2018.

[2] M. Rosenfeld, "Analysis of hashrate-based double spending," *arXiv preprint arXiv:1402.2009*, 2014.

[3] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network.," in *USENIX Security Symposium*, pp. 129–144, 2015.

[4] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International conference on financial cryptography and data security*, pp. 436–454, Springer, 2014.

[5] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *Security and Privacy (SP), 2017 IEEE Symposium on*, pp. 375–392, IEEE, 2017.

[6] M. Apostolaki and L. Vanbever, "Sabre: Protecting bitcoin against routing attacks." submitted.

[7] "Fast Internet Bitcoin Relay Engine." https://www.falcon-net.org.

[8] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[9] "Protocol documentation." https://en.bitcoin.it/w/index.php?title=Protocol_documentation&oldid=65104, March 2018.

[10] *select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing*, 15.09.17 ed., September 2017. Linux man page select(2).

[11] J. Weidendorfer, "kcachegrind - callgraphviewer." https://kcachegrind.github.io/, April 2013.

[12] S. Ghemawat, "Cpu profiler." https://gperftools.github.io/gperftools/cpuprofile.html, May 2018.

[13] H. Ware and F. Frederick, *vmstat - Report virtual memory statistics*, 3.3.12 ed., September 2011. Linux man page vmstat(1) from procps-ng.

[14] *top - display Linux processes*, 3.3.12 ed., July 2016. Linux man page top(1) from procps-ng.

[15] B. Lankester, M. K. Johnson, M. Shields, C. Blake, D. Mossberger-Tang, and A. Cahalan, *ps - report a snapshot of the current processes.*, 3.3.12 ed., August 2015. Linux man page ps(1) from procps-ng.

[16] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *AUUG Conference Proceedings*, vol. 85, AUUG, Inc., 2002.

[17] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan, "The implications of shared data synchronization techniques on multi-core energy efficiency.," in *HotPower*, 2012.

[18] The Bitcoin Core developers, "Bitcoin core integration/staging tree." https://github.com/bitcoin/bitcoin, 2018. ref:bf3353de90598f08a68d966c50b57ceaeb5b5d96.