UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Marek Pagel

Performance Testing Bulletin Board Implementations for Online Voting

Bachelor's Thesis (9 ECTS)

Supervisor: Sven Heiberg, MSc
Supervisor: Janno Siim, MSc

Tartu 2017

Performance Testing Bulletin Board Implementations for Online Voting

Abstract:
Online voting is an electronic voting method in which the process of casting a vote is done using the Internet as its communication medium. One component of some online voting systems is a public bulletin board (PBB), used to provide election transparency and correctness verifiability. PBB is a component for publishing data in a way that makes modifying or deleting already published data very difficult without leaving evidence of such actions. The security and liveness of this component has to be ensured. This means that implementing PBB is a machine replication problem at its core with some specific requirements inherited from the context of online voting. This work takes a look at two software solutions that can be used for such purpose and analyses their performance in testing environment imitating real election workload.

Keywords: Public Bulletin Board, Online Voting, Distributed System, State Machine Replication, Performance Testing

CERCS: P170 Computer science, numerical analysis, systems, control

Teadetetahvlite jõudlustestimine internetivalimiste kontekstis

Lühikokkuvõte:
Internetihääletamine (i-hääletamine) on hääletamisviis, mille puhul hääl liigub valija seadmest urni Interneti vahendusel. I-hääletamise süsteemide sagedaseks komponendiks on avalik teadetetahvel, kuhu registreeritud andmete abil tagatakse valimiste läbipaistvus ja auditeeritavus. Avalik teadetetahvel on komponent, mis võimaldab registreerida andmeid viisil, mis muudab nende hilisema muutmise või kustutamise keeruliseks. Teadetetahvli puhul on oluline teenuse tagatud kättesaadavus ja toimimine. Sellest tulenevalt on avaliku teadetetahvli implementeerimise tuumküsimuseks korrektne masinate kordistamine, mille muudab keerulisemaks i-hääletamise spetsiifilised lisanõuded. Selle töö käigus käsitletakse kahte olemasolevat tarkvaralahendust, mida saab kasutada teadetetahvli realiseerimisel, uurides nende jõudlust testkeskkonnas, mis imiteerib pärisvalimiste töökoormust.

Võtmesõnad: Avalik teadetetahvel, i-hääletamine, Hajussüsteem, Olekumasina kordistamine, Jõudlustestimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction

Electronic systems have steadily been replacing, changing and modernizing more traditional nondigital process models on individual and governmental scales. One such traditional process is the process of elections, in which votes are most commonly cast on paper. Online voting is becoming a viable and more widespread alternative to the paper voting. For example, online voting has been used in Estonia for more than 10 years with the proportion of votes cast electronically rising steadily. Now more governments are looking into the technology in hopes of reducing costs, reaching remote voters and speeding up the process. This means online voting systems must be ready to accommodate the increasing number of voters.

Bulletin board is one of the possible components in an online voting system. The main idea behind the concept of bulletin board is to provide a way to publish some data and be assured that the data can't be deleted or modified at a later time without leaving evidence of such actions to the public. Its purpose is to ensure that the electronic ballot box is consistent and correct throughout the election in a way that provides election verifiability for third parties. As such, the security and liveness of the bulletin board has to be guaranteed throughout the election period. This can be achieved with machine replication, but due to inherited complexity of online voting, most traditional replication methods are not suitable for this task.

Goal of this work is to introduce and test two readily available software solutions for this use case. In order to successfully carry out the performance testing, proper testing environment has to be defined and set up. Finally, the test results are assessed in the context of system requirements and usability in online voting.

The first section of this work introduces the software solutions and their communication protocols. The second section describes the testing environment, various tools and the setup used in this work. The third section presents and analyses the performance results gathered during the testing.

2 Tested PBB Implementations

This section gives an overview of the software solutions to be tested with a focus on the communication protocols involved in processing new requests.

2.1 Peered Bulletin Board

Bulletin board described in [CS14] was developed as part of the vVote system for Victorian State election 2014 [BCH+12]. The Victorian State election has a two-week-long early voting period followed by the election day.

The bulletin board consists of robust system of multiple peers, all of which receive items, provide receipts and publish information. Throughout the day peers might have different internal states but synchronization protocol is ran at the end of the day. This is done at an off peak time (to ensure liveness) right before publishing. Each day is considered one period. This system is stated to work correctly under the assumption that a threshold of peers is honest and operational at any given time. The threshold t required to achieve correctness must be greater than two-thirds of the total number n of peers: $t > 2n/3$. As long as a threshold of peers remains operational the system tolerates communication failures, individual peer failures and even malicious peers.

The article states that the key properties required of this implementation are:

- only items that have been posted to the bulletin board may appear on it;
- any item that has a receipt issued must appear on the published bulletin board;
- two clashing items must not both appear on the bulletin board;
- items cannot be removed from the bulletin board once they are published.

The bulletin board has two main protocols: one for posting items and the other for publishing the bulletin board, which consists of an optimistic scenario and a fallback protocol. This system is developed in Java programming language.

2.1.1 Posting and Acknowledgment

To post an item x in period p on the bulletin board the following steps are taken:

1. Client sends x to each peer;
2. Each peer checks that x doesn't clash with any previous posts;
3. Each peer signs (p, x) and sends the result to every other peer;

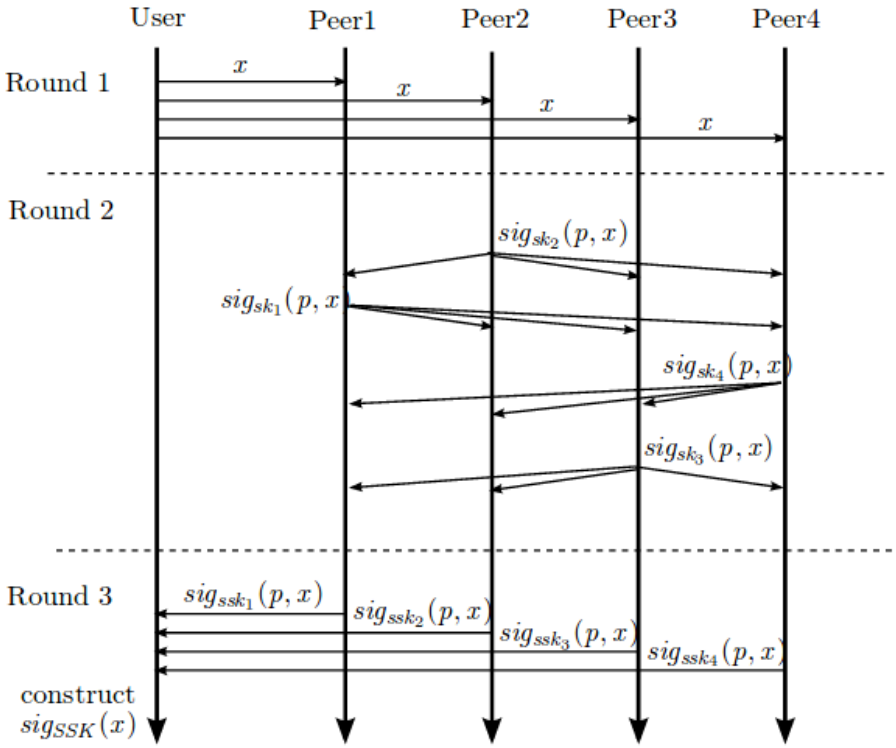

Figure 1. Posting Protocol [CS14]

4. When peer has received threshold of signatures (including its own) it sends its share of threshold signature on (p, x) back to client

5. Client combines threshold of signature parts into signature and verifies it.

Threshold signature scheme is a protocol that allows any subset of K parties out of L to generate a cryptographic signature equivalent to more traditional cryptographic signature, but disallows creation of such signature if fewer than K parties participate in the protocol. K and L are integers fixed before key generation protocol. [Sho00]

All necessary persistent data, including client messages and different signatures, is kept in a local database. This protocol is visualized on Figure 1. $Sig_{sk}(\cdot)$ denotes generation of traditional cryptographic signature with key sk, while $sig_{ssk}(\cdot)$ denotes generation of cryptographic signature share with key ssk.

2.1.2 Publishing the Bulletin Board

The bulletin board is published at the end of the period. Before it can be published a threshold of the peers must agree on the final version of the bulletin board. Peer i's

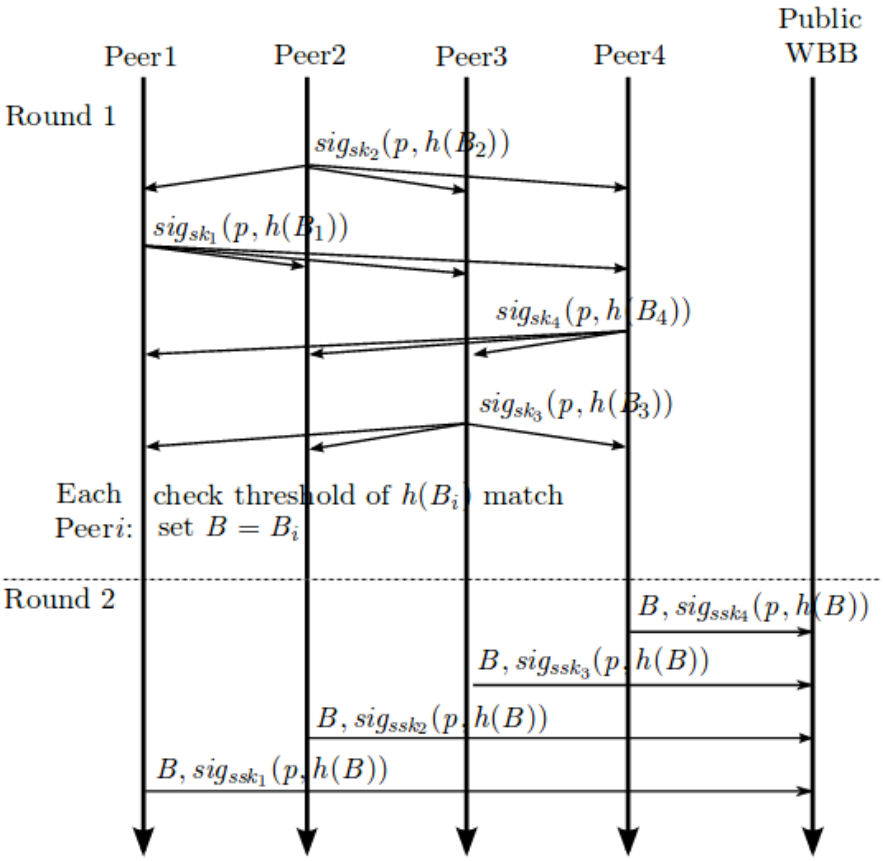

Figure 2. Optimistic Protocol [CS14]

local record of bulletin board $B_{i,p}$ consists of items it has received a threshold number of signatures on. Peers try to run an optimistic protocol first and follow up with a fallback protocol if necessary.

The optimistic protocol will succeed if at least a threshold of peers were properly working throughout the period and consequently have all the items to be published. The peers each sign the hash of their local bulletin board and broadcast it. If a threshold agrees on the hash then they can issue the bulletin board and their part of threshold signature to the publicly visible bulletin board service. This protocol is visualized on Figure 2. $H(\cdot)$ denotes cryptographic hash function.

If the optimistic protocol does not produce consensus, then the fallback protocol is ran at least once. This involves peers sharing their bulletin board information with each other. Each peer broadcasts its database of signatures $D_{i,p}$ and update their own database with missing signatures from other peers' databases. They then recalculate their bulletin board and run the optimistic protocol again. Under some reasonable live-

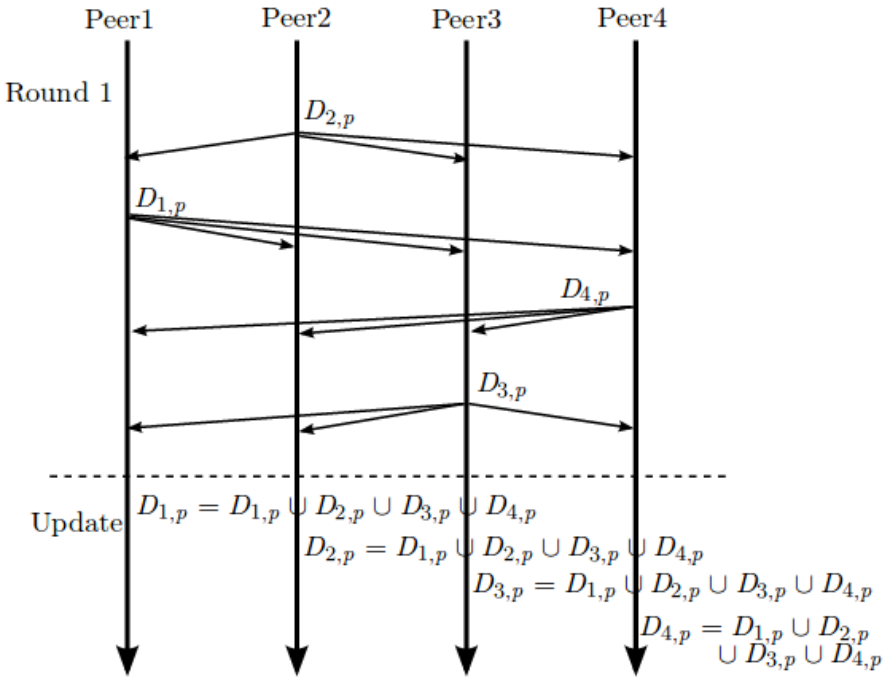

Figure 3. Fallback Protocol [CS14]

ness assumptions, only one round of fallback protocol is ever needed. This protocol is visualized on Figure 3.

2.2 BFT-SMaRt

BFT-SMaRt (**B**yzantine **F**ault-**T**olerant **S**tate **M**achine **R**eplication) is a robust Java-based BFT SMR library which aims to be high-performance and simple to use dependency for other services and protocols to build upon. [BSA14]

In this work a very simple service was built upon the BFT-SMaRt library. The service is accepting client requests and stores them in a MongoDB instance. Such service is too simple for any real life application, but the main goal is to test the underlying replication protocol.

Byzantine fault tolerance characterizes distributed system's tolerance against class of failures named after Byzantine General's Problem, first described in [LSP82]. Byzantine fault is a fault that presents itself with different symptoms to different observers. In practice, this means an arbitrary deviation from expected behavior, which can be caused by system failure, connection delays or even malicious activity.[DHSZ03] Furthermore, Byzantine fault tolerance is becoming increasingly important on one hand due to growing reliance of industry and government online services and on the other hand due to

the growth in size and complexity of software. As a result, system failures of any kind are becoming less acceptable, but eliminating their possibility is becoming more difficult.[CL+99]

State machine replication is a popular replication method that enables a set of peers to execute the same sequence of operations for a service even if a number of them are faulty. The core of state machine replication is the consensus problem, which is how to come to an agreement on one result in a distributed system. [SB12]

Consensus algorithm used in BFT-SMaRt is an extension of the leader-driven Byzantine consensus algorithm described in [Cac09], which in turn is based on the very famous Paxos consensus algorithm first introduced in [Lam98]. The system tolerates f faulty nodes in a n-node system where $n \geq 3f + 1$. [BSA14]. This means both vVote and BFT-SMaRt both tolerate the same number of faulty nodes in a n-node system.

The communication protocol of BFT-SMaRt involves more rounds and messages compared to the bulletin board described before. On the other hand, practical optimizations could lead to better overall performance. One such optimization is handling multiple client requests in a batch.

To post an item x on bulletin board the following steps are taken:

1. Client sends x to each peer;
2. Elected leader proposes the value to all peers by sending them PROPOSE message (n, x) where n is the consensus index;
3. Each peer checks that they are able to accept the value;
4. Each peer sends WRITE message $(n, H(x))$ to every other peer;
5. Upon receiving threshold number of WRITE messages, each peer sends ACCEPT message $(n, H(x)$ to ever other peer;
6. Upon receiving threshold number of ACCEPT messages, value is accepted
7. Accepted value is processed and response is sent to the client

Checking whether a peer is able to accept a value is not actually very relevant in properly working system. Leader is proposing values with monotonically increasing indexes and it is required for a peer to decline the proposal, if it has already accepted a value for given index. This becomes relevant if more than one peer considers themselves the leader and they start sending out competing proposals.

Leader election in this system is solved in a very simple manner. Namely, each peer is given an integer identification number and the peer with the smallest id among reachable replicas is considered to be the leader. Under normal conditions, leader never has to change.[BSA14]

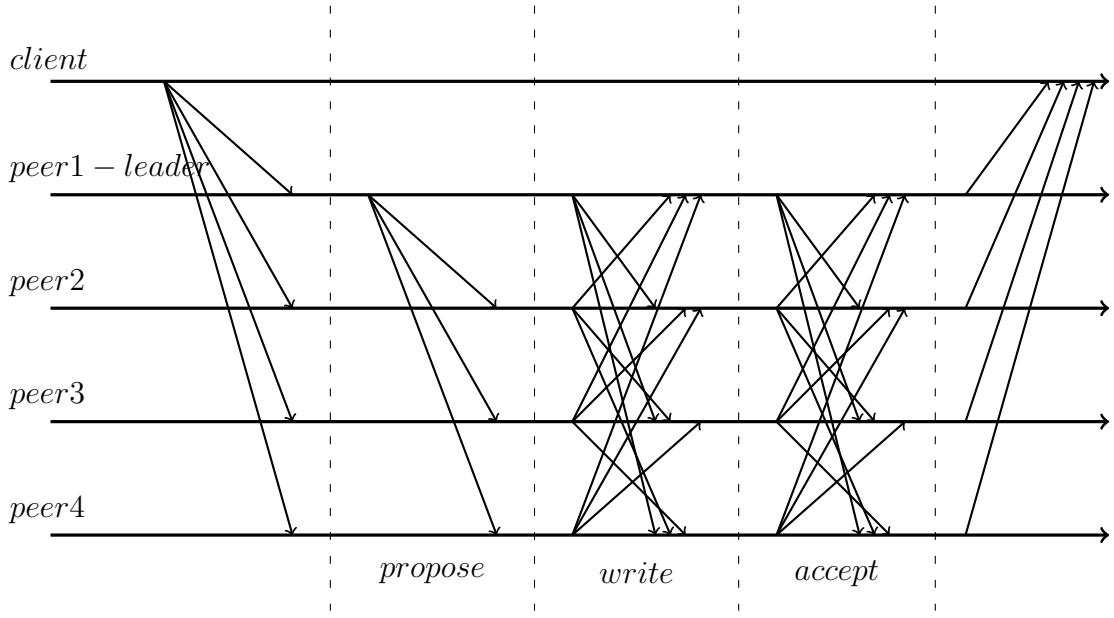

Figure 4. Byzantine Consensus

This protocol is visualized on Figure 4. Source code and more info about the library can be found at [bft].

vCPU count	4
Memory	16 GiB
Physical Processor	Intel Xeon E5-2676 v3 with a footnote "may launch on an Intel Xeon E5-2686 v4 (Broadwell) Processor"
Clock Speed	2.4 GHz

Figure 5. m4.xlarge system statistics

3 Testing Setup

Services are set up on Amazon Web Services (AWS) platform.[AWS] All nodes of a system are ran on the same system, eliminating possibly significant network latency and simplifying system load observations. M4.xlarge instances are used as they are the smallest instances with high network performance and subjectively reasonable hardware specifications for the task. M4 instances are defined as general purpose instances that provide a balance of computation, memory and network resources. See Figure 5 for more detailed m4.xlarge machine definition based on AWS homepage information. The same instance class is used for both server and client side to assure that client machine is capable of generating enough load on the server.

Required machines are the following:

- Server machine - used to host the system under testing;
- Client machine - used to stress the system under testing;
- Collection machine - used to collect and visualize performance data.

Both implementations are written in the programming language Java, so neither can have a significant edge in that regard. For all code and configuration files used to conduct the testing see appendix.

3.1 Testing-Related Software

In addition to the software of the systems to be tested, multiple other software solutions are required to make the testing environment easier to setup and to collect necessary performance data.

3.1.1 Docker

Docker[1] is an open-source project that automates application deployment inside isolated containers. Unlike machine virtualization, containers do not include full operating

[1] `https://www.docker.com/`

system, but only the bare minimums required for each particular application. This technology allows for easier system setup and more importantly clear resource isolation for accurate data collection. In addition we use Docker-Compose tool for easier management of multi-container systems. Docker is required in every machine.

Using Docker requires creation of the Dockerfile, which tells Docker how to build and run our container. In addition docker-compose configuration file is required to configure and launch multiple containers at once.

3.1.2 cAdvisor

cAdvisor[2] is used to gather data about resource usage and performance characteristics of running containers. It has native support for Docker containers. All the collected data is exported to the collection machine, in order minimize performance impact on the server machine. cAdvisor is required in the server machine.

3.1.3 InfluxDB

InfluxDB[3] is a time-series database. As cAdvisor only displays real-time information, InfluxDB is required to store that information in order to analyze data over longer time range. InfluxDB is required in the collection machine.

3.1.4 Grafana

Grafana[4] is an open-source tool for data querying, visualization and analysis. This tool is used to query InfluxDB and export visualized performance results. Grafana is required in the collection machine.

3.2 Testing Process

Systems were tested with periodically increasing load. Test started with load of 60 requests per minute and the number of requests doubled every five minutes. There were ten phases in total, which means the final load was 30720 requests per minute or 512 requests per second. The testing limit was loosely derived from Estonian Online Voting statistics. According to [vvk] in the 2015 Riigikogu Elections the most online votes cast in a single day was 39039. If making the assumption that most of the votes were cast during a period of 10 hours, it comes to average load of $39039/10/60 \simeq 65$ votes per minute. This gives the starting point of the load configuration. The upper limit of load is more arbitrary, but given the fact that a system under such load could accept a little

[2] `https://github.com/google/cadvisor`
[3] `https://www.influxdata.com/products/open-source/#influxdb`
[4] `https://grafana.com/`

over 300 million votes in a 7 day period, it is very likely well above requirements for most elections.

4 Testing Results

Testing results are presented by stating the overall outcome of the test, followed by more detailed analysis of different system resource based on Grafana produced graphs where applicable. The graphs feature resource usage of all peers separately on one graph. Sum of resource usage of all containers could be more readable, but it may also hide possible differences between each container.

It should be noted that although AWS m4.xlarge instance according to Figure 5 has clock speed of 2.4 GHz, the real maximum system load is clock speed times vCPU count, which in this case is 9.6 GHz.

4.1 vVote PBB

According to manual found in vVote documentation repository[5] the system is rated at 80 votes per second on benchmark system. It is unclear how benchmark system is defined. During tests conducted here, no such performance was achieved.

During the first test, around 20 minutes into the process, a sudden drop in CPU usage can be noted, as seen on Figure 6. Upon taking a look at system logs, it shows that at around the same time, large amount of timed out requests start to appear. On the other hand, Figure 6 does not indicate significant amount of CPU usage.

RAM usage, as seen on Figure 7, is also relatively small compared to total system memory availability. Network usage is insignificant in amount and couldn't possibly be the limiting factor. Network usage is quite similar to BFT-SMaRt's network usage in the respective phases, if not even a little smaller.

The test was carried out for a second time while manually monitoring system resource usage directly on the server machine in real-time. This monitoring indicated system to be under 100% CPU load around 20 minutes into to process and the logs confirmed request timeouts again. Yet, collection machine still showed insignificant CPU usage. RAM usage was reported similarly on server and benchmark machines. A third test was conducted, this time all 4 peers were started in the same container, to try a different setup and further remove possible network related issues. Still, exactly the same behavior was observed in every aspect. One possible explanation to this phenomenon could be that cAdvisor was unable to properly work under loaded system.

All three tests have very similar time of failure - the start of fifth phase, in which 16 votes per second are transmitted. By the time of failure, peers had successfully accepted little over 5000 votes, which is an expected number considering the timeline. Derived from these tests, system is rated at somewhere between 8-16 votes per second. It can also be stated that the system is primarily CPU intensive.

[5] `https://bitbucket.org/vvote/doco/src`

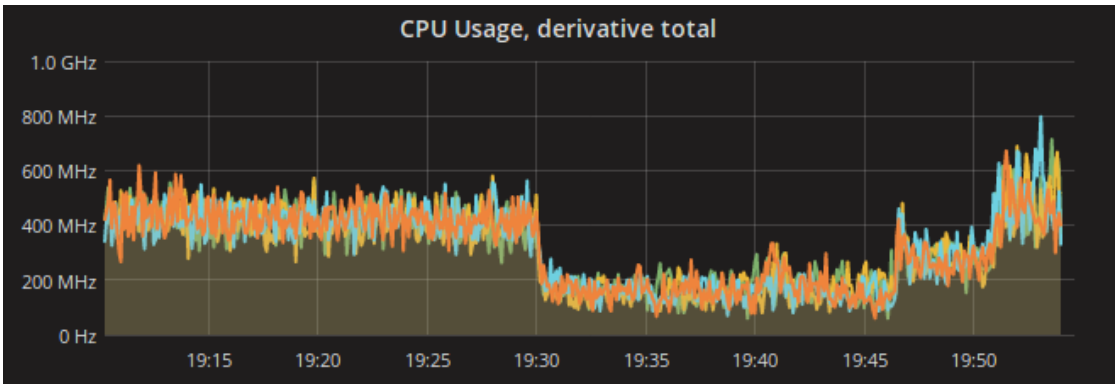

Figure 6. vVote CPU usage graph

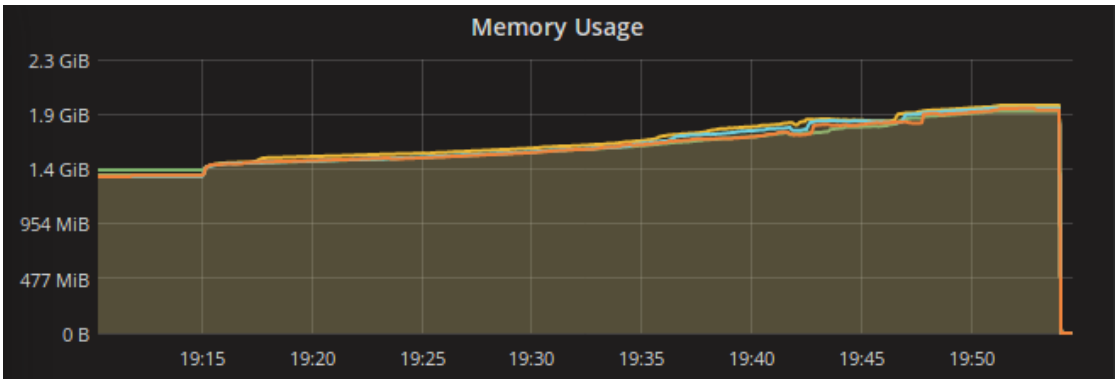

Figure 7. vVote RAM usage graph

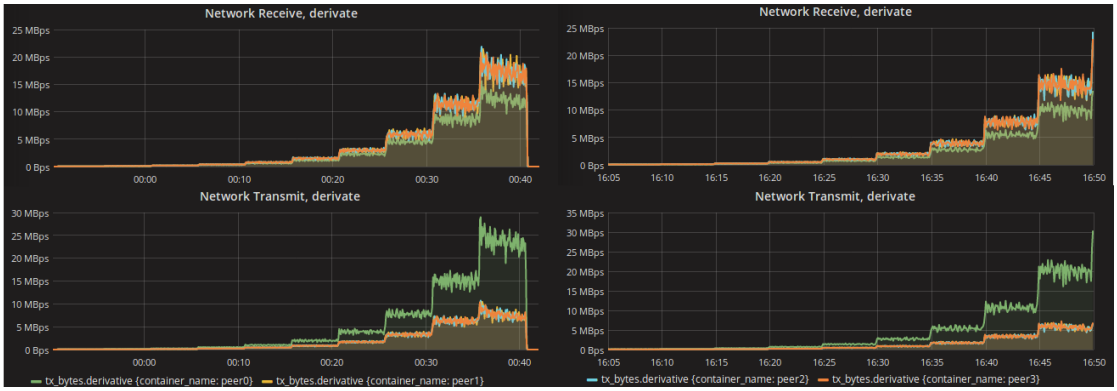

Figure 8. BFT-SMaRt network usage graph. Test 1 on the left and Test 2 on the right.

Large discrepancy between expected and resulting performance leaves possibility for further work to more closely study the underlying causes for such discrepancy. This could involve more direct contact and collaboration with original system authors.

4.2 BFT-SMaRt

BFT-SMaRt successfully managed to work properly throughout the test. To further study the system, another test was ran with twice the client request size. This imitates the increase in the cryptographic key size used to encrypt ballots. The second test did not finish successfully as one of the containers crashed due to java virtual machine running out of heap space. This was caused by the system running out of memory.

The graphs included in the analysis are taken from the first test as the results between the two tests are very similar. Exception to this is the network graph which differs significantly between the two tests. The graphs include the whole period of the test, which is 10 phases or 50 minutes for the first one and 9 phases or 45 minutes for the second.

Figure 8 represents network usage graph. There is one peer standing out by having clearly different network usage compared to the rest. This is due to Peer0 being the consensus leader, who is responsible of transmitting the PROPOSE messages as shown in Figure 4. Consequently the leader has higher transmission rate but lower receiving rate. Even further, it can be seen that the amount of data the leader peer has transmitted more than non-leader peers equals the amount of data non-leader peers as a group have received more than the leader.

Network usage difference between the two tests is evident. If comparing the respective phases, test 2 uses an average of about 25% more network resources. The peak network bandwidth usage of single peer was about 30 MBps.

CPU usage of containers is visible on Figure 9. First of all, it is clear that CPU

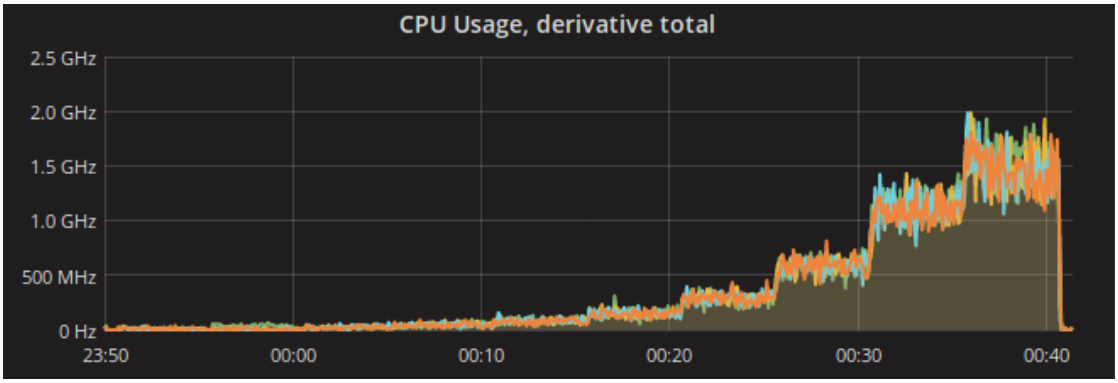

Figure 9. BFT-SMaRt CPU usage graph

usage is in strong correlation with the number of incoming requests. In general, it could be said that doubling the load causes CPU usage to double. Interestingly enough, the transition final phase shows a smaller jump in CPU usage than earlier phases. The peak CPU usage of single container was about 2 GHz and the peak system load was $2GHz * 4/9GHz \simeq 83\%$.

RAM usage of containers is visible on Figure 10. BFT-SMaRt system baseline RAM usage is very low compared to Vvote system's. RAM usage does start to grow significantly in phase 8 and reaches close to system limits in the very end of the test. Given the fact that one of the containers crashed in test 2 due to lack of memory, it is clear that RAM becomes the system bottleneck if load increases indefinitely. BFT-SMaRt holds copy of vote database in memory for snapshotting and peer restoring purposes. This could explain as to why RAM usage increased faster in test 2 with bigger vote sizes. To see how big of an impact this snapshotting bears on RAM usage, it is required to calculate its size. At the end of test 1, each peer had received 306931 votes, each vote about 750 bytes in size. This means the total size of the in-memory snapshots was $4 * 306931 * 750 \simeq 920MB$. In test two, at the time of crash each peer had received 189435 votes, each vote about 1500 bytes in size. This gives the respective size of $4 * 189435 * 1500 \simeq 1140MB$. Calculations confirm that higher RAM usage in test 2 is at least partially caused by larger vote files. Still, snapshot consumes relatively small percentage of total RAM.

It is clear that BFT-SMaRt has potential to be used as a public bulletin board. Implementation of server side application for this project was most likely too simple for any real life applications, but it gives a clear starting point. According to [CS14], trying to reach consensus every time an item is posted creates too much of an overhead. It may be true in their protocol, but it seems to be doable in general.

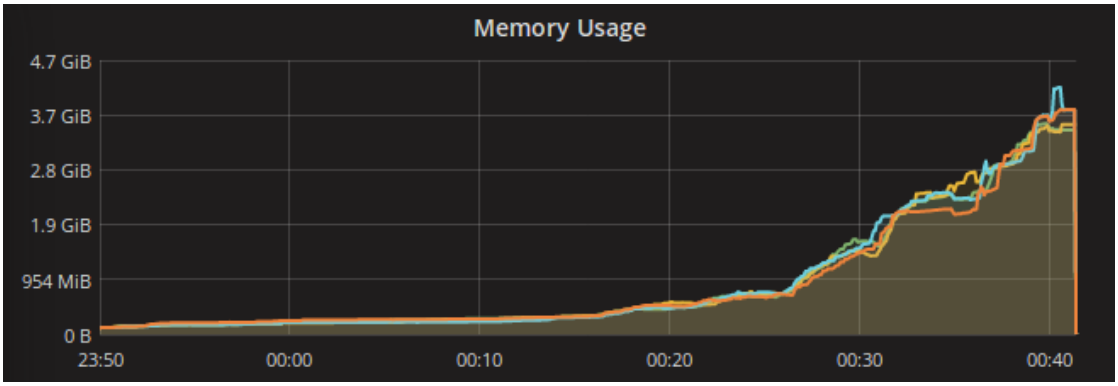

Figure 10. BFT-SMaRt RAM usage graph

5 Conclusion

The aim of present work was to find feasible software solutions for use as a public bulletin board in electronic voting systems. The approach to measuring such feasibility was testing the performance of readily available software implementation as opposed to studying the theoretical performance characteristics of underlying algorithms. Two such software solutions were chosen: one part of a complete voting system actually used in practice and another more general consensus library that could be used as a core for more specific system. Testing environment and scenario were defined and necessary tools configured. Finally the systems' hardware resource usage was monitored under periodically increasing constant load and the results studied.

The robust peered bulletin board used in the vVote system showed unexpectedly low performance compared to statements from system's manual. Requests started to timeout under load of 16 requests per second, rating the system between 8 to 16 requests per second. There were also issues with collection of performance statistics during the test that persisted over multiple test attempts. Because of this, results can't be considered conclusive and future work could include collaboration with original vVote developers in an attempt to find out the cause of low performance observed in this work.

The Byzantine fault tolerant state machine replication library BFT-SMaRt showed very promising results. The application had quite well rounded hardware usage statistics with memory being the limiting resource for the machine instance used in this work. The application managed to handle client requests all throughout the test, rating the system at least at 512 requests per second. As the main goal was to test the performance of underlying consensus algorithm implementation, this performance might not be achievable in a production ready system. Future work could theoretically or practically try to find out how much overhead has to be added to the system for it to be production ready and how it would affect the performance.

Future work could also test the performance of the systems under different kind of load (e.g. spike load) or even expand the testing onto software solutions not handled in this work.

References

[AWS] Elastic compute cloud(ec2) - cloud server & hosting - aws. `https://aws.amazon.com/ec2/`. Visited: 25.04.2017.

[BCH+12] Craig Burton, Chris Culnane, James Heather, Thea Peacock, Peter Y. A. Ryan, Steve Schneider, Sriramkrishnan Srinivasan, Vanessa Teague, Roland Wen, and Zhe Xia. Using prêt à voter in victorian state elections. In *Proceedings of the 2012 International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2012.

[bft] Bft-smart by bft-smart. `http://bft-smart.github.io/library/`. Visited: 25.04.2017.

[BSA14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[Cac09] Christian Cachin. Yet another visit to paxos. *IBM Research, Zurich, Switzerland, Tech. Rep. RZ3754*, 2009.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[CS14] Chris Culnane and Steve Schneider. A peered bulletin board for robust use in verifiable voting systems. *CoRR*, abs/1401.4151, 2014.

[DHSZ03] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[SB12] J. Sousa and A. Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48, May 2012.

[Sho00] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000*, pages 207–220. Springer, 2000.

[vvk] E-hääletamise statistika - valimiste arhiiv - valimised. http://www.vvk.ee/arhiiv/e-statistika/. Visited: 25.04.2017.

Appendix

I. Repository

All code and configuration files used in this work are available at `https://github.com/marekpagel/bc_thesis_code`

II. Testing Instructions

The following is a guide for reproducing the testing process used in this work. File paths referenced in this guide are relative paths from the root directory of the code repository found in Appendix I. The cAdvisor-InfluxDB-Grafana setup is inspired by a web article by Brian Christner available here.

Setting Up AWS Instances

It is required to create and start 3 instances of the following types:

- m4.xlarge with Ubuntu 16.04 Operating System as the server machine;
- m4.xlarge with Ubuntu 16.04 Operating System as the client machine;
- m4.large with Ubuntu 16.04 Operating System as the collection machine.

After starting the machines, note the public IP addresses for the server and collection machines.

Detailed guide for this process can be found at AWS homepage.

Installing Docker and Docker-Compose

It is required to install Docker on server and collection machines. In addition, Docker-Compose has to be installed on server machine. Instructions for installing Docker can be found here. Instruction for installing Docker-Compose can be found here.

Configuring Collection Machine

1. Create SSH connection to the collection machine.
2. Clone the project's code repository:

```
$ git clone https://github.com/marekpagel/bc_thesis_code
```

3. Start Influxdb instance with the following docker command:

```
$ docker run -d -p 8083:8083 -p 8086:8086 --expose 8090 --expose
    8099 -e PRE_CREATE_DB=cadvisor --name influxsrv tutum/
   influxdb:latest
```

4. Start Grafana instance with the following docker command:

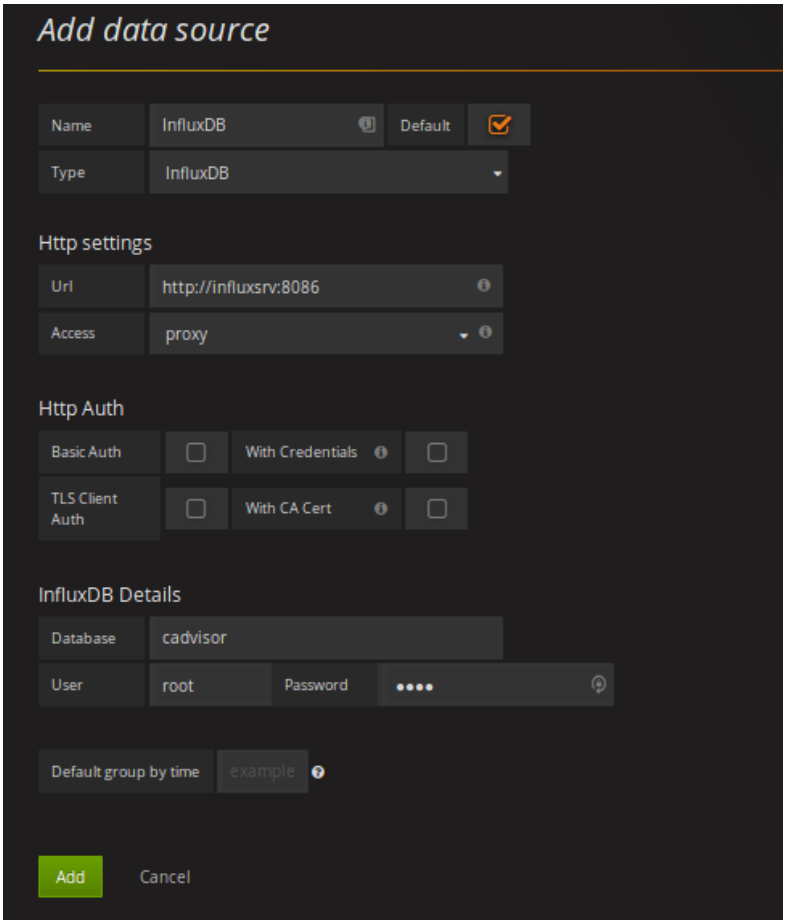

Figure 11. Grafana InfluxDB configuration

```
$ docker run -d -p 3000:3000 -e HTTP_USER=admin -e HTTP_PASS=
    admin -e INFLUXDB_HOST=localhost -e INFLUXDB_PORT=8086 -e
    INFLUXDB_NAME=cadvisor -e INFLUXDB_USER=root -e INFLUXDB_PASS
    =root --link=influxsrv:influxsrv --name grafana grafana/
    grafana:latest
```

5. Go to <collection-machine-IP>:3000 with a webbrowser.
6. Log in with username and password 'admin'.
7. Add new Data Source as shown in Figure 11. InfluxDB password is 'root'.
8. Import dashboard from `grafana/benchmark.json`.

Configuring Server Machine

1. Create SSH connection to the server machine
2. Clone the project's code repository with submodules:

```
$ git clone --recursive https://github.com/marekpagel/
    bc_thesis_code
```

3. Open `/etc/default/grub` with root access and modify the following line:

```
GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0"
```

   to

```
GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0
    cgroup_enable=memory"
```

4. Run the following command:

```
$ sudo update-grub
```

5. Reboot the instance.
6. Start cAdvisor instance with the following docker command. Make sure the correct collection machine IP address is inserted:

```
$ docker run --volume=/:/rootfs:ro --volume=/var/run:/var/run:rw
     --volume=/sys:/sys:ro --volume=/var/lib/docker/:/var/lib/
    docker:ro --publish=8080:8080 --detach=true --name=cadvisor
    google/cadvisor:latest -storage_driver=influxdb -
    storage_driver_db=cadvisor -storage_driver_host=<collection-
    machine-ip>:8086
```

Configuring Client Machine

1. Create SSH connection to the server machine
2. Clone the project's code repository with submodules

```
$ git clone --recursive https://github.com/marekpagel/
    bc_thesis_code
```

3. Install Java8 JRE

```
$ sudo apt-get update
$ sudo apt-get install openjdk-8-jre-headless
```

Testing vVote

1. Create SSH connection to the server machine
2. Modify `vvote/suvote_wbb/wbb/release_demo/Peer1/wbbconfig.json` so the json list with key "peers" contains correct server machine IP addresses.
3. Repeat the process for configuration files of Peers 2-4.
4. Start the service with the following commands:

```
$ cd vvote/suvote_wbb/
$ docker-compose up -d
```

5. Create SSH connection to the client machine
6. Modify `vvote/my_app_client/MBBConfig.json` to contain the correct server machine IP addresses.
7. Start the client with the following commands:

```
$ cd vvote/my_app_client/
$ java -jar out/suvote_client.jar
```

8. Test ends automatically in 50 minutes

Testing BFT-SMaRt

1. Create SSH connection to the server machine
2. Start the service with the following commands:

```
$ cd BFT_SMART/
$ docker-compose up -d
```

3. Create SSH connection to the client machine
4. Modify `BFT_SMART/my_app/config/hosts.config` to contain the correct server machine IP addresses.
5. Start the client with the following commands:

```
$ cd BFT_SMART/my_app/
$ java -cp out/* pagel.thesis.Client
```

6. Test ends automatically in 50 minutes

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Marek Pagel**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis

   Performance Testing Bulletin Board Implementations for Online Voting

   supervised by Sven Heiberg and Janno Siim

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 11.05.2017