Throughput Analysis of a Trustchain Protocol Implementation

Tudor Chirila
Supervisors: Johan Pouwelse, Bulat Nasrulin
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Tudor Chirila
Final project course: CSE3000 Research Project
Thesis committee: Johan Pouwelse, Bulat Nasrulin, Koen Langendoen

Abstract

Blockchain technologies offer decentralized, secure, and transparent transaction systems but face significant scalability challenges, especially in mobile and peer-to-peer environments. Directed Acyclic Graph (DAG)-based protocols, such as Trustchain, present a promising alternative by allowing each agent to maintain a personal chain of relevant transactions, thereby improving throughput and reducing resource consumption. Despite its theoretical potential, Trustchain has not yet been effectively implemented for smartphones. To evaluate performance, we provide an implementation over two communication protocols, focusing on throughput. We found that our QUIC-based implementation displays an over three times throughput improvement when keeping a QUIC connection open, and can confidently store at least 28 536-byte blocks per second. However, our UDP-based version outperforms our implementation, confidently storing at least 500 668-byte blocks per second.

1 Introduction

Blockchain technologies have begun to attract wide-spread interest after Satoshi Nakamoto's Bitcoin paper was published in 2008[1]. The idea took hold so quickly - and the core philosophy driving their widespread adoption - is the promise of secure data storage through a network of agents with equal rights, as opposed to depending on a central authority.

Despite their increased adoption, there is a significant limitation that widely used technologies, such as Bitcoin, cannot scale for more general purpose applications as the network size grows: Bitcoin can process only about seven transactions per second and require confirmation times of around ten minutes[2]. This by design bottleneck has sparked interest and research towards a new class of blockchain systems: DAG (Directed Acyclic Graph)-based [3], which, unlike the linear blockchain, has the underlying structure of a Directed Acyclic Graph, promising higher scalability.

In this paper, we will focus our attention on the Trustchain protocol [4] - a DAG-based blockchain protocol that aims to be Sybil-resistant - a type of attack where an attacker can generate multiple identities in the network to gain disproportionate influence.

Smartphones now outnumber traditional computers and are an important internet access-point. A native mobile Trustchain client would unlock the possibility of multiple new technologies, such as a more scalable system of transactions. However, there is a unique set of challenges when developing such a system:

1. There are limited computing resources on a mobile phone.
2. Although there are testing frameworks, many are not mobile-oriented.
3. There are use-cases where throughput is critical, such as retail payments.

To the best of our knowledge, no successful native Trustchain smartphone application exists to date, even more so for analyzing and optimizing its throughput. We aim to bridge this gap by developing a prototype, as inspired by the published draft [5]. Due to time constraints, we focus on the core blockchain logic - creating a message, integrating it as a payload, sending it to be signed by another agent, then storing the block upon receipt, without the consensus algorithm described in the section. Furthermore, we are measuring the throughput a 2-peer network the Trustchain application supports, and we investigate possible ways of further optimizing for this benchmark. In short, we aim to answer the following research questions:

1. How can we build a smartphone application which implements the core features of the Trustchain protocol?

2. Given the base implementation of the application, what is its throughput performance (that is, the number of transactions per second that the app will be able to support)?

The rest of the paper has the following structure: we will present our research methodology in Section 2, followed by an overview of our Trustchain implementation in Section 3. In Section 4, we will delve deeper in the experimental analysis, and into the incremental findings we have observed regarding throughput in our 2-peer experiments. Section 5 covers elements of responsible research, and we mark the end by limitations, conclusions and future work in Section 6.

2 Methodology

2.1 Prototype development

The aim of the first part of the project is developing a viable Trustchain implementation. In extension, its criteria needs to be:

- Implement a main version with a mature framework used for peer-to-peer communication.
- Connect to another peer.
- Be able to create a block and sign it.
- Attach a message to a block and send it to another peer. Both the sending agent and the receiving agent need to add it to their own block.
- Implement alternative networking protocols versions alongside the main one, such as UDP[6] and TFTP[7].

We will deem our application as ready-to-be-benchmarked for the second step.

2.2 Throughput analysis

Having a core application that integrates the main Trustchain features - that is, wrapping the payload inside a block, signing it, sending it to another agent that then signs and thus completes, then sends it back to us - we will now perform an iterative experimental analysis on how the throughput gets impacted in our application under diverse circumstances. First of all, it is essential to define throughput in our application. By definition, throughput is the number of work a system can handle over a specific period. In order to define "handled work", we can consider multiple definitions. However, the most relevant ones are at a few keypoints in our workflow which allow as having a block being "processed":

- getting the block signed by both agents.
- getting the block back right after being signed by the receiving agent.
- getting the block stored on the sending agent.

For completeness and ease of implementation, we will consider the last two alternatives as variants of a block being processed, with a strong preference for the latter - since that is a stage when a block completely finishes its completeness and is stored on the chain. Moreover, we will decide to keep some factors as controlled as possible, while others we need to keep controllable when running a benchmarking experiment. The controlled factors include:

- Running the experiments on the same private Wi-Fi, with no other devices connected to the network.
- Running on the same model of smartphones, with the same Android version.

On the other hand, the ones we can control for ourselves are:

- The duration of the simulation (in ms).
- the approximate number of messages we will be sending per second (by which we divide 1000ms to obtain the interval we are waiting between messages).
- the payload size (in bytes).

Ideally, as in other benchmarking works [8], we will increase the workload - that is, the number of messages we are sending per second - exponentially, in order to reach the point where it faster bottlenecks. We will thus fix the same duration - long enough to minimize initial connection establishments -, exponentially increase the workload (the number of messages we are sending per second), and choose arbitrary payload sizes in order to measure the throughput as the workload increases. We will put both the `iroh` and `udp` versions under the same loads which will exponentially rise, under arbitrary payload sizes.

3 Overview of Trustchain Communication Implementations

3.1 Design & Implementation

The elements described in this section have been collaboratively developed withing our research group. Finally, each of us delved into analyzing different aspects: energy efficiency, latency, robustness, storage and throughput.

The goal of the first part of this research is building a peer-to-peer application that is using the Trustchain framework - in particular, that is able to send messages to a peer and respect the core feature of the Trustchain protocol: creating a block proposal with the given payload, sign it, send it to another peer, then receive it signed back and adding it to our own chain.

Technology evaluation We have first started with our peer-to-peer networking layer decision. In order to build our application, we have investigated 3 options of already-existing libraries for peer finding and optimal message sending for smartphone agents:

- ipv8 - python library that performs peer identity and NAT puncturing.
- Kotlin-ipv8 - a Kotlin-native version of ipv8 with limited functionality.
- iroh / rust-iroh - a mature Rust framework that uses QUIC[9] and a relay network to find and send messages between peers.

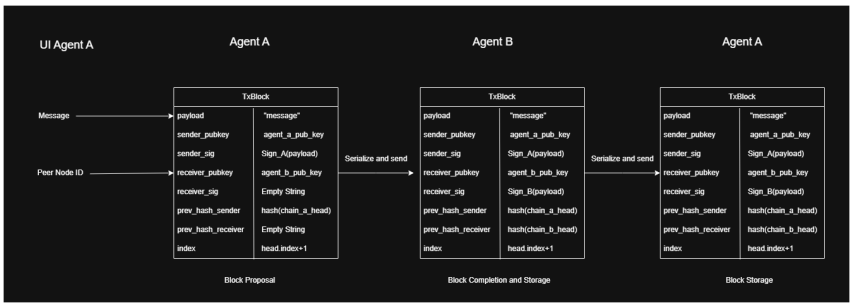

Figure 1: Blockchain implementation workflow

Taking into account the fact that we want a general-purpose application and the maturity of the technologies, we have finally decided upon using a Core **Rust** layer for implementing the Trustchain logic - block creation, signing, sending, as well as networking with `iroh`, and a thin layer of **Kotlin** for user interaction and for testing the application on Android devices, the bridge between the two being the **Java Native Interface (JNI)**. We have further thought about interoperability between Rust and Swift, thus strengthening our decision of decoupling the core Trustchain and networking logic from the device's operating system and future adoption on iOS devices.

Moreover, we will compare our results with a significantly lighter variant for networking - that is, raw UDP, and provide insight towards the possibility of adopting a solution for raw UDP instead of a heavy framework such as iroh - with the caveat that UDP does not provide the reliability that QUIC does, and only works on a local network peer.

Trustchain Architecture Implementation Having taken strong inspiration from the IETF Trustchain draft [5], we have come up with the workflow for creating a valid block, attaching a payload and sending it to another client presented in Figure 1, using the block structure in Figure 2. It includes a stage of block proposal creation (where the agent signs the payload with its private key) then a completion and storage phase by the receiving agent (where it signs it as well and stores it, then sends it back to the original sender), followed by the original sending agent storing the completed block. We will further port this blockchain implementation to our network-specific versions.

Considering its core features that are shared with the draft description, such as having each agent signing the message and publishing its public key, as well as the core philosophy of keeping an independent chain where each of the agents are, we have assumed this version to be a viable implementation for the purpose of measuring its throughput.

3.2 QUIC-based Implementation (`iroh`)

This implementation leverages the underlying implementation of QUIC in rust, through the `quinn` library. It provides out-of-the-box peer-to-peer message transmission API's, including asymmetric encryption (which is used for generating the private-public key pair for each agent), as well as peer-discovery techniques. QUIC is a modern transport protocol designed for multiplexed, low-latency, and reliable communication, making it well-suited for

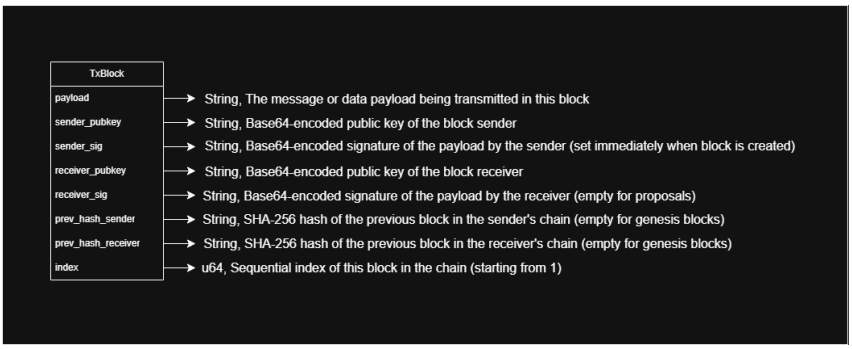

Figure 2: Transaction Block Structure

peer-to-peer networking in resource-constrained environments. In this implementation, each smartphone agent operates a QUIC endpoint, handling incoming and outgoing connections through bi-directional streams. The Rust code exposes JNI interfaces for Android integration, enabling the mobile app to start listeners, send messages, and retrieve benchmarking results. We have implemented two important functions using rust-iroh. The first one is highlighting the receiving logic for rust:

```
start native OS level thread for listening to incoming messages
create new tokio runtime
inside this tokio runtime, create a QUIC endpoint through ‘quinn‘’s
    Endpoint interface
loop {
    wait for incoming connection
    if received, spawn a new green (user-level) thread. inside this thread:
    {
        await for incoming connection to convert into a valid ‘Connection‘
    object
        when this happens, open a send_stream and a recv_stream through
    the ‘.accept_bi()‘ method
        read from receiving stream the message and
        process_it. This is the point where the blockchain implementation
    takes over.
        wait for stream to finish
        close connection.
    }
}
```

Another critical function is, on the other hand, the one for sending a message, which illustrates the exact same behavior, with the change of opening a connection to another peer instead of waiting for an incoming one, and using the sending stream instead of the receiving stream.

3.3 Raw UDP Implementation (udp)

Our UDP implementation is rather trivial in nature. We are leveraging the `std::net` rust crate, specifically the UDP interface. We are simply opening a known UDP socket and sending a message block. As far as our receiving logic goes, we have a separate thread with a loop on which we continuously scan for messages on our own socket. However, for the scope of this research, this implementation is good enough: we aim to showcase a comparison between a complex framework (iroh, which uses QUIC) and the simple UDP transmission of blocks, knowing their main differences, which are reliability, and UDP being only usable on the local network due to its lack of complex peer-finding protocols.

By implementing and comparing these two approaches, we are able to experimentally analyze the impact of protocol choice on throughput, efficiency, and reliability for decentralized, DAG-based blockchains on smartphones. This comparative analysis provides practical guidance for future Trustchain deployments in mobile and resource-constrained environments.

4 Experimental setup and results discussion

4.1 Experimental Setup

We have developed a simulation run implementation which, over a user-specified duration, sends messages with a user-defined size at a user-defined workload (messages per second rate). We are recording timestamps in a shared data structure, which include the message itself - the full block in a serialized format - as well as the operation type, which describes the operation we are executing - in our case, sending a message (`send_benchmark_sc`), processing it right after it has arrived from our peer (`processing_message`) and right after the completed block has been finally stored in the sending agent (`store_block_e`). After this duration ends, we are exporting the benchmark result in csv format and based on the count of the rows of a specific operations (sent, or stored), we can derive the specific throughput.

Using our implementation, we have leveraged Android Studio's native option of flashing the application into our available phones. We conducted our experiments using two identical Motorola Moto G04s smartphones connected to the same Wi-Fi network, with specifications written in Appendix C. The experimental workflow for iroh involved the following steps:

1. Open the application on Agent B and click on `GO TO IROH COMMUNICATION`.
2. On Agent B, click the 'Copy' button next to the third textbox, which includes Agent B's NodeId.
3. Paste the NodeId to a shared Google Docs file.
4. Open the shared Google Docs file on Agent A , copy the NodeId.
5. Open the application on Agent A and click on `GO TO IROH COMMUNICATION`.
6. Paste the NodeId as a destination, in the first text field.
7. Fill in the `Simulation Config` text box with its respective message size, messages per second and simulation test duration values.

8. Click on `Run Simulation` and wait for the csv filed export message to be displayed on the screen, remember the name.
9. Kill the app on Agent B.
10. Kill the app on Agent A.
11. Take the csv file from Agent A's storage through a cable and using Android Studio's `Device explorer`.

Regarding the UDP version, we take a slightly different approach, due to its lack of overhead for connections and rather simple implementation, which allows the application to perform in a similar manner over time.

1. Hardcode a set of workloads we want to run this implementation for. There will be an automatic 20 seconds timeout between each experiment, to make sure that no packets from the previous iteration have interfered with the current simulation run.
2. Open the application on Agent B and click on `GO TO UDP COMMUNICATION`.
3. On Agent B, scroll to the bottom of the page and manually copy the displayed NodeId.
4. Paste the NodeId to a shared Google Docs file.
5. Open the shared Google Docs file on Agent A, and copy the NodeId.
6. Open the application on Agent A and click on `GO TO UDP COMMUNICATION`.
7. Paste the NodeId as a destination, in the first text field.
8. Fill in the `Simulation Config` text box with its respective message size, messages per second and simulation test duration values.
9. Click on `Run Simulation` and wait for the csv file export message to be displayed on the screen.
10. Kill the app on Agent B.
11. Kill the app on Agent A.
12. Import the csv file from Agent A's storage using Android Studio's `Device explorer`.

4.2 Iterative Runs and Results

4.2.1 Rust Iroh with constant reconnection - 10 Byte payload

First of all, as mentioned in the methodology, we will select an arbitrary workload and perform a test. We will arbitrarily choose 10 Bytes as the message size - which, although rarely used in practice - will give us a strong insight into the behavior of the throughput of the 2-peer system.

We are running for the workloads [1, 2, 4, 8, 16, 32, 64, 128]. However, when running with the 64 workload, we notice something interesting: the application breaks with a memory-related exception, and we no longer have enough results to make confident assumptions. We have theorized the root of this issue to be the heap getting overflown by `tokio`'s green

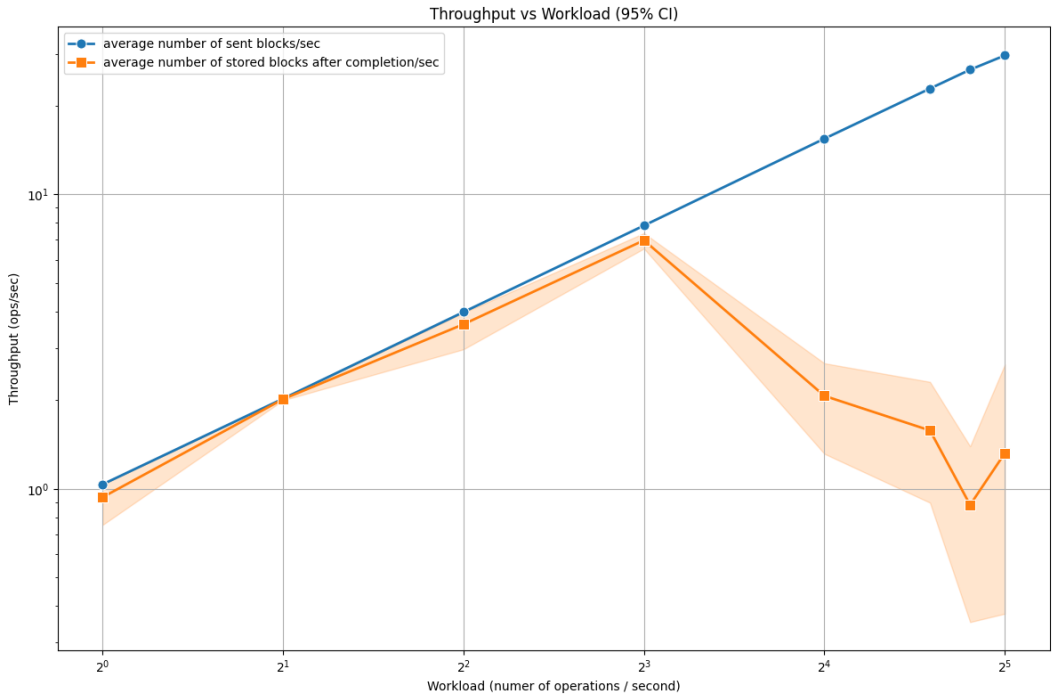

Figure 3: Sent and stored throughput, 10B payload, 25 seconds, iroh with reconnection

threads generated with Rust's `tokio::spawn`, however, a more in-depth analysis needs to be performed in this regard. The method described in subsection 4.2.4 fixes this issue.

Therefore, in order to obtain meaningful results, we run the benchmark with the following workloads: [1, 2, 4, 8, 16, 20, 24, 28, 32, 36] messages / 1000 ms, for 5 times each workload for 25 seconds, and obtain the result in Figure 3.

We notice a peak in the store throughput at around 8 operations per second, where it reaches (according to the Results in Appendix A) a mean of 7.84 ± 0.009 ops/sec. The send throughput grows naturally with the actual sending instructions we do in the simulations, which we expected. The throughput increasing together with the sending rate, then plummeting/decreasing is expected as well. However, we notice quite a large variance from the point of 16 ops/second onward - which indicate less reliable results as the workload increases. However, they do indicate that they do not overcome the aforementioned peak.

4.2.2 Rust Iroh with constant reconnection - 100 Byte payload

In order to display the behavior of the store throughput over a larger payload size, we have chosen 100 Bytes and kept the rest of the conditions the same. We have obtained the result in Figure 4.

We again notice a peak in the store throughput at around 8 operations per second, where it reaches a mean of 7.02 ± 0.469 ops/sec. The send throughput grows naturally with the actual sending instructions we do in the simulations, which we expected. The throughput increasing together with the sending rate, then plummeting/decreasing is expected as well. There is an increase in variance in the latter workloads - however, based on their 95%

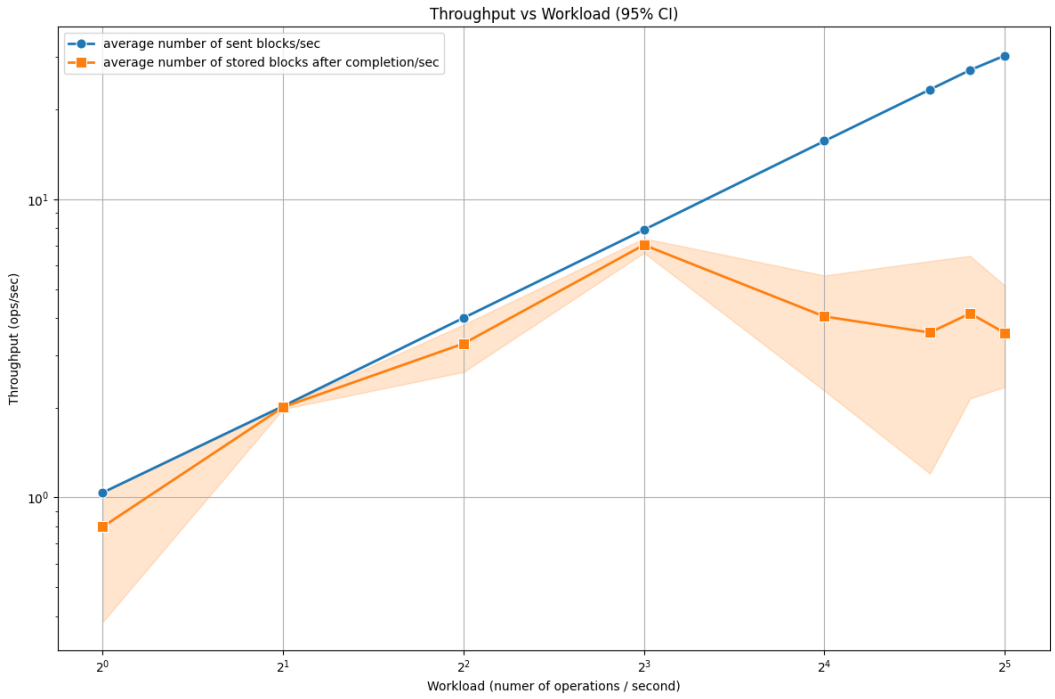

Figure 4: Sent and stored throughput, 100B payload, 25 seconds, iroh with reconnection

Confidence interval, even its upper bound doesn't seem to surpass the mentioned peak.

4.2.3 Raw UDP - 100 Bytes

Running the UDP version provides us with outstanding results: the sending throughput and store throughput match for the workloads [1, 2, 4, 8, 16, 24, 28, 32], which we have also tested in the iroh version. Since presenting the graph would be redundant, we provide the following rates:

```
Workloads: [1, 2, 4, 8, 16, 24, 28, 32]
Send counts (avg): [25.0, 50.0, 99.0, 196.0, 387.4, 576.0, 670.6, 756.4]
Store counts (avg): [25.0, 50.0, 99.0, 196.0, 387.4, 576.0, 670.6, 756.2]
```

However, this raises the question: under which conditions does the UDP store throughput start to diverge from the sending throughput? We will delve deeper in this aspect in the following parts.

4.2.4 Rust Iroh connection optimization - 100 Bytes

The rust-iroh implementation breaking at around 40 messages per second imposes a critical limitation. Moreover, in our current implementation for iroh (as also presented in the pseudocode in 3.2), we are waiting for an Incoming QUIC connection **inside the loop**, then closing it each time. While this is a generally good idea and supports messages to different peers, in our current use case - that is, testing for only two peers - we can perform

an optimization which is in tune with QUIC's spirit of being able to multiplex between multiple streams.

In order to reflect a use-case where there is a long connection between two-peers, we are going to perform the following variation from the original pseudocode, which we have reflected in this implementation:

```
create new tokio runtime
inside this tokio runtime, create a QUIC endpoint through ‘quinn‘’s
    Endpoint interface
wait for first incoming connection
once it has arrived and turned into a valid ‘Connection object‘ do
    loop {
        open a send_stream and a recv_stream through the ‘.accept_bi()‘
    method
            read from receiving stream the message and
            process_it. This is the point where the blockchain
    implementation takes over.
            wait for stream to finish
        }
}
```

Now, running the same workloads as in section 4.2.2, we obtain the result in figure 5.

This is an unexpected improvement. As opposed to the store peak being around 8 ops/second (of 7.84 ± 0.009), we have now obtained a stable peak at 28 operations/second, which according to the results in the annex, corresponds to 25.58 ± 0.100 ops/sec. We have therefore managed to obtain a maximum store throughput **3.26 times larger** than iroh with the optimization of keeping the connection open.

Moreover, we have also increased the capacity of the application itself. The system no longer breaks, and we are able to put iroh under a load of 1000 messages/second without any issues, as opposed to it breaking at 40 messages/second.

4.2.5 Limit testing: iroh versus UDP - 128 Bytes

Now that we have enhanced iroh's performance for large workloads, we are free to test its capabilities, up to 1024 messages/second and put iroh and UDP to a more fair comparison.

We are making a small change in the code, as we are now dividing 1000 ms by the workload - which, due to integer division and Kotlin working with integer number of milliseconds - will not provide us with fair results. We are going to put both the iroh and the udp-version under the workloads of [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024], over the course of 1024 ms batches, for an overall of 25 seconds, as before.

We have obtained the result in Figure 6. Although we have obtained a maximum throughput when sending approximately 527.72 per second (that is, with a 95% confidence interval, 295.96 ± 236.577), a more reliable throughput rate can be recognized for the workload of 32 messages per second: with a 95% confidence interval, 28.87 ± 0.04 blocks per second. Analyzing the payload size that we have recorded in our benchmarks (attached in Appendix B), we see that there is a minimum of 536 characters in the message size, which corresponds to 536 Bytes/message. Therefore, we can confidently say that we have obtained at least 15474.32 ± 21.44 Bytes/second of storage throughput.

However, when we compare the result with what we obtain in UDP, we notice how in these specific circumstances, iroh falls rather short.

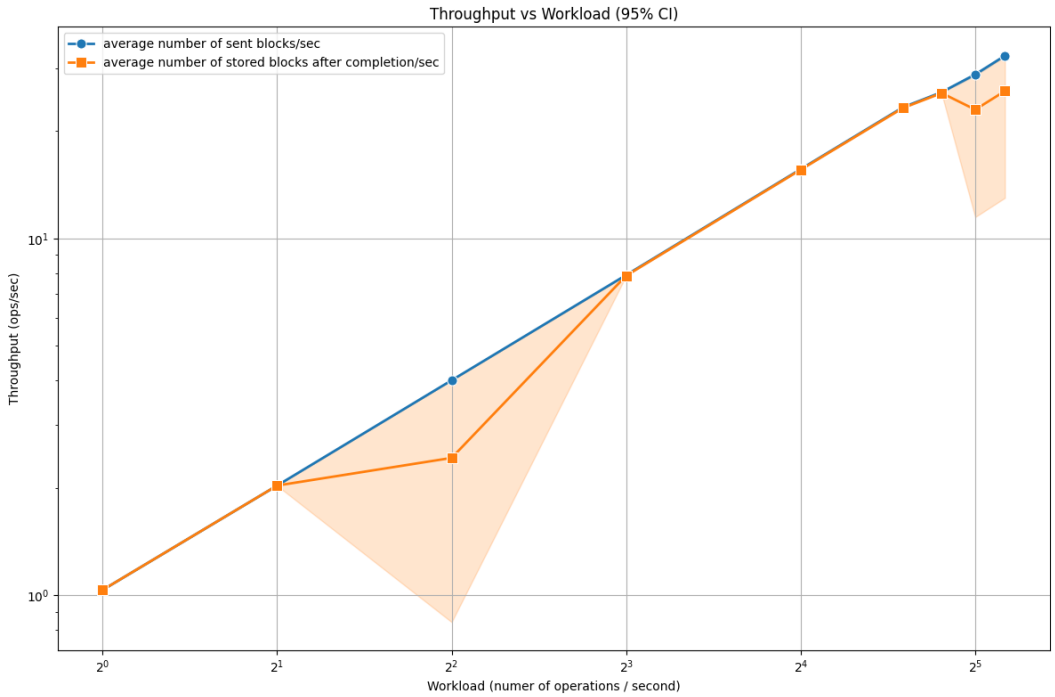

Figure 5: Sent and stored throughput, 100B payload, 25 seconds, iroh with reconnection

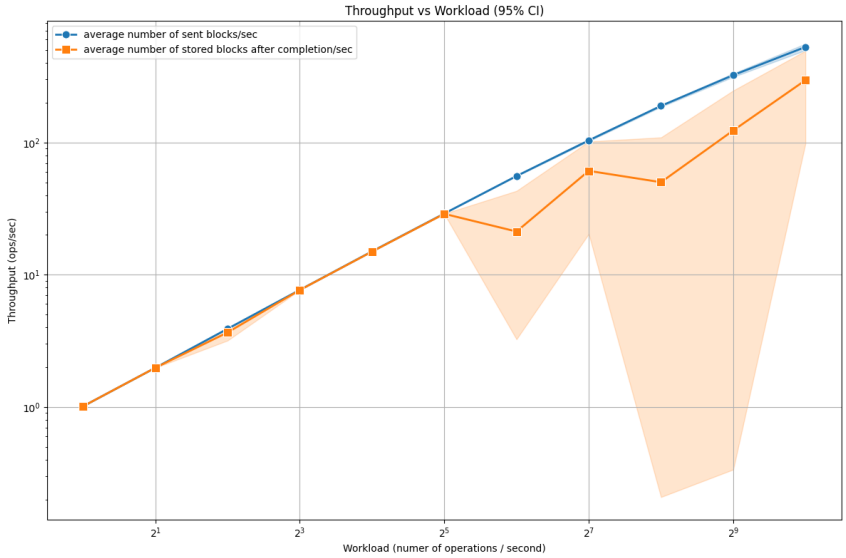

Figure 6: Sent and stored throughput, 128B payload, 25 seconds, iroh with connection keeping

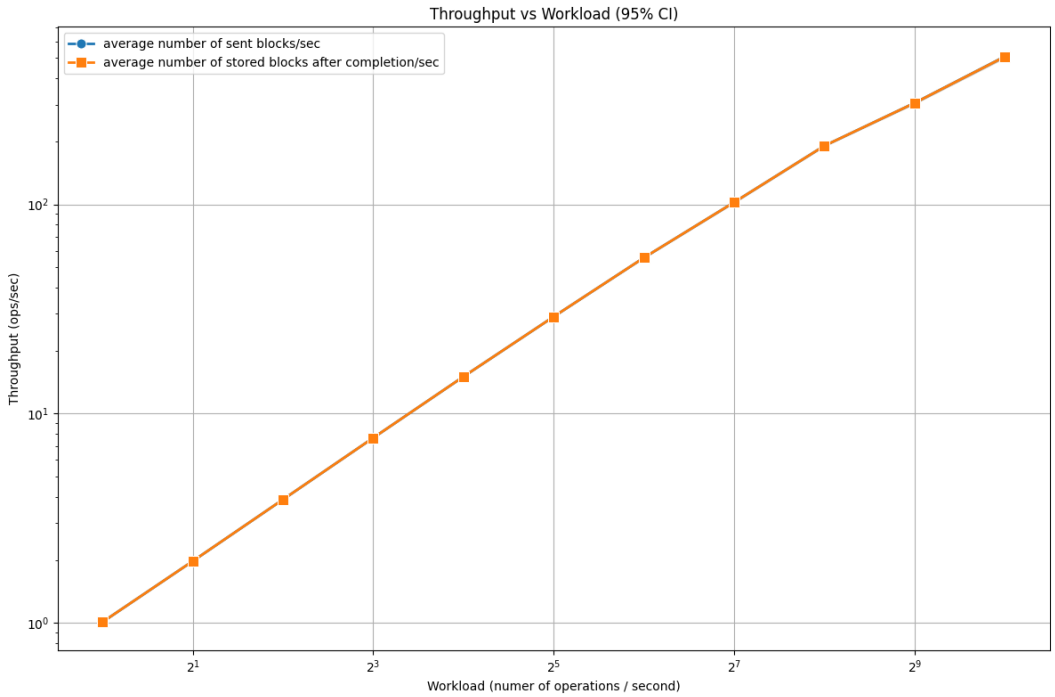

Figure 7: Sent and stored throughput, 128B payload, 25 seconds, UDP version

The sending and the receiving throughput are so close that the difference between them can be barely seen on the graph in Figure 7. Taking the three last 95% CI intervals, we have the following results:

```
Workload 256:
  Send: 190.57 ± 0.754 ops/sec
  Store: 190.53 ± 0.754 ops/sec
Workload 512:
  Send: 305.32 ± 3.895 ops/sec
  Store: 305.26 ± 3.909 ops/sec
Workload 1024:
  Send: 508.99 ± 8.156 ops/sec
  Store: 508.27 ± 7.283 ops/sec
```

Taking into consideration that the error is almost 64 times lower than the mean, we can safely say that we have reliably obtained at least 500 messages/second throughput in one of our best instances. Taking the measured block size (Annex B, point 4) of at least 668 bytes, we have obtained a total of 333400 Bytes/second of message-storing throughput, which is around 326KB / second.

4.2.6 Final insights

The main insight is that keeping a QUIC connection open, instead of closing it and reopening again is a strong feature of the protocol, both intuitively and as observed in our experiments.

Moreover, it has enhanced the throughput of the two-peer blockchain system of over three times. Another one is that the viability of our blockchain implementation is confirmed by the results of our experiments. In extension, we have obtained expected results in the case of throughput: constantly rising with the same rate as the sending one, followed by a point where the sending throughput and the storage throughput diverge.

5 Responsible Research

We have used publicly available information and methods in providing our blockchain implementation and results.

Moreover, we have detailed each result and possible shortcomings we are aware of as well as provided the code through which we have obtained these results.

Our implementation of the IETF Trustchain protocol inherently considers user privacy, as each agent maintains its own blockchain containing only personally relevant transactions. Moreover, we have used dummy data as payload for our implementation - thus, no sensitive information has been used in our application.

In regards to reproducibility, we do believe that us providing the models of the smartphones we have used, as well as their full specification, the code, and detailed instructions over how to run the experiments makes the overall work highly reproducible - at least to the extent of noticing the same key behaviours of the throughput. There, is however a factor that has an impact over full reproducibility, which is depending on the Wi-fi network. Despite the fact that we have aimed to limit this factor as much as possible - that is, by running all our experiments on a private network, with no other devices - this is of course a factor that is far beyond our control. Moreover, we do depend on external devices - that is, the two smartphones - which might also impose some issues, such as possible manufacturing issues, which are also far beyond our knowledge. In regards with LLM usage, we have mainly leveraged it for converting natural language into latex (example: pseudocodes used in this paper) and for highlighting grammar mistakes.

6 Limitations, conclusions and future work

6.1 Limitations

One of our main challenges include the premise of this work - that is, our implementation of the Trustchain protocol. Although we find that we have implemented a viable version for measuring its throughput, a deeper analysis needs to be pursued over a more accurate implementation of the Trustchain protocol - respecting the exact block structure and including the consensus algorithm.

Another challenge which highlights the limitations of the current research involves the lack of multiple physical agents we can experiment on. For a more proper investigation, various network topologies need to be analyzed.

One other possible drawback of our implementation is marked by the benchmarking overhead: although our current way offers significant data, such as the timestamp, the block structure and the message content, we could have registered only the count of the stored blocks in the specific timeframe. However, we would then lack the confidence that the message structure is fully correct. Moreover, we have redundantly benchmarked the processing time, possibly adding more overhead.

Another problem we acknowledge in this research is the low amount of times we run the experiment for a specific workload. A large cause to this is the unfeasibility of manually executing the iroh experiment at scale, due to the high involvement of human intervention. We will therefore need a way to automatize this process.

Lastly, we will thoroughly need to extend code quality and robustness. Although it gave us enough insight for measuring purposes, we cannot ignore some of its observed faults over time: on our rust iroh optimization, it seems to sometime disconnect for larger workloads, which gets reflected in the displayed graph. Moreover, in the original iroh version, thorough investigations need to be conducted over the reason of it breaking at around 40 messages / second.

6.2 Conclusions

In conclusion, we have successfully developed a smartphone application leveraging mature peer-to-peer technologies facilitated by the rust-iroh framework, implementing a viable Trustchain version for testing purposes. Thus, we have answered the first research question by developing an application that creates a block, sends a payload, then receives back the completed block and stores it in its own chain.

Moreover, we have benchmarked the storage throughput and have gained relevant insights. First of all, we noticed that keeping a QUIC connection open in `iroh` can offer over three times the throughput in our recorded experimental data. Moreover, our implementation can confidently process around 28 blocks/second with a 128 Bytes payload over a sending rate of 29 messages per second, marking a maximum processing throughput of around 15KB/second, having answered our second research question, and providing an optimization from the baseline, being able to increase the stable storage throughput rate by over three times. However, our UDP-based version outperforms our implementation, confidently storing at least 500 668-byte blocks per second.

6.3 Future work

Our future work focuses on improving the limitations this paper proposes. First of all, we aim to adopt a more accurate Trustchain representation, which also includes the consensus algorithm. We aim to write more formal unit tests for the correctness of the protocol, as well as following a more structured software engineering approach. Moreover, we will further investigate issues that the implementation currently poses and fix them.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008. Whitepaper.

[2] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[3] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. Sok: Diving into dag-based blockchain systems, 2022.

[4] Pim Otte, Martijn de Vos, and Johan Pouwelse. Trustchain: A sybil-resistant scalable blockchain. *Future Generation Computer Systems*, 107:770–780, 2020.

[5] Johan Pouwelse. Trustchain protocol. Internet-Draft, IETF. draft-pouwelse-trustchain-01, June 5 2018.

[6] User Datagram Protocol. RFC 768, August 1980.

[7] Dr. Karen R. Sollins. The TFTP Protocol (Revision 2). RFC 1350, July 1992.

[8] Bulat Nasrulin, Martijn de Vos, Georgy Ishmaev, and Johan Pouwelse. Gromit: Benchmarking the Performance and Scalability of Blockchain Systems. In L. O'Conner, editor, *Proceedings of the 4th IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS 2022)*, pages 56–63. IEEE, 2022.

[9] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

A Confidence intervals - experiments 4.2.1 - 4.2.4

1. Confidence intervals deviations for the experiment described in 4.2.1, Figure 3. Throughput Summary with 95

```
Workload 1:
  Send: 1.04 ± 0.001 ops/sec
  Store: 0.94 ± 0.175 ops/sec
Workload 2:
  Send: 2.02 ± 0.001 ops/sec
  Store: 2.01 ± 0.017 ops/sec
Workload 4:
  Send: 3.98 ± 0.005 ops/sec
  Store: 3.62 ± 0.630 ops/sec
Workload 8:
  Send: 7.84 ± 0.009 ops/sec
  Store: 6.98 ± 0.491 ops/sec
Workload 16:
  Send: 15.46 ± 0.024 ops/sec
  Store: 2.07 ± 0.745 ops/sec
Workload 24:
  Send: 22.87 ± 0.029 ops/sec
  Store: 1.58 ± 0.815 ops/sec
Workload 28:
  Send: 26.51 ± 0.066 ops/sec
  Store: 0.88 ± 0.568 ops/sec
Workload 32:
  Send: 29.64 ± 0.069 ops/sec
  Store: 1.32 ± 1.320 ops/sec
```

2. Confidence intervals for the experiment described in 4.2.2, Figure 4.

```
Throughput Summary with 95% CI:
Workload 1:
  Send: 1.04 ± 0.001 ops/sec
  Store: 0.80 ± 0.395 ops/sec
Workload 2:
  Send: 2.02 ± 0.007 ops/sec
  Store: 2.01 ± 0.031 ops/sec
Workload 4:
  Send: 4.00 ± 0.006 ops/sec
  Store: 3.28 ± 0.685 ops/sec
Workload 8:
  Send: 7.90 ± 0.009 ops/sec
  Store: 7.02 ± 0.469 ops/sec
Workload 16:
  Send: 15.67 ± 0.018 ops/sec
  Store: 4.04 ± 1.969 ops/sec
Workload 24:
  Send: 23.32 ± 0.059 ops/sec
  Store: 3.57 ± 2.822 ops/sec
Workload 28:
  Send: 27.10 ± 0.068 ops/sec
  Store: 4.14 ± 2.279 ops/sec
Workload 32:
  Send: 30.32 ± 0.093 ops/sec
  Store: 3.56 ± 1.552 ops/sec
```

3. Confidence intervals for the experiment described in 4.2.4, Figure 5.

```
Throughput Summary with 95% CI:
Workload 1:
  Send: 1.03 ± 0.001 ops/sec
  Store: 1.03 ± 0.001 ops/sec
Workload 2:
  Send: 2.03 ± 0.002 ops/sec
  Store: 2.03 ± 0.002 ops/sec
Workload 4:
  Send: 4.01 ± 0.001 ops/sec
  Store: 2.43 ± 1.894 ops/sec
Workload 8:
  Send: 7.90 ± 0.005 ops/sec
  Store: 7.87 ± 0.020 ops/sec
Workload 16:
  Send: 15.65 ± 0.005 ops/sec
  Store: 15.59 ± 0.017 ops/sec
Workload 24:
  Send: 23.32 ± 0.029 ops/sec
```

```
  Store: 23.21 ± 0.032 ops/sec
Workload 28:
  Send: 25.72 ± 0.084 ops/sec
  Store: 25.58 ± 0.100 ops/sec
Workload 32:
  Send: 28.83 ± 0.257 ops/sec
  Store: 22.98 ± 11.205 ops/sec
Workload 36:
  Send: 32.54 ± 0.142 ops/sec
  Store: 25.90 ± 12.632 ops/sec
```

B iroh results for 128 Bytes, connection optimization - experiment 4.2.5

1. Confidence intervals for throughputs - iroh version

```
Throughput Summary with 95% CI:
Workload 1:
  Send: 1.01 ± 0.002 ops/sec
  Store: 1.01 ± 0.002 ops/sec
Workload 2:
  Send: 1.98 ± 0.003 ops/sec
  Store: 1.98 ± 0.003 ops/sec
Workload 4:
  Send: 3.90 ± 0.001 ops/sec
  Store: 3.66 ± 0.472 ops/sec
Workload 8:
  Send: 7.68 ± 0.002 ops/sec
  Store: 7.66 ± 0.019 ops/sec
Workload 16:
  Send: 15.02 ± 0.006 ops/sec
  Store: 14.96 ± 0.021 ops/sec
Workload 32:
  Send: 29.02 ± 0.013 ops/sec
  Store: 28.87 ± 0.045 ops/sec
Workload 64:
  Send: 55.78 ± 0.419 ops/sec
  Store: 21.22 ± 20.793 ops/sec
Workload 128:
  Send: 103.75 ± 1.826 ops/sec
  Store: 61.03 ± 48.766 ops/sec
Workload 256:
  Send: 189.22 ± 3.452 ops/sec
  Store: 50.08 ± 69.495 ops/sec
Workload 512:
  Send: 323.25 ± 11.415 ops/sec
```

```
  Store: 123.28 ± 147.558 ops/sec
Workload 1024:
  Send: 527.71 ± 35.360 ops/sec
  Store: 295.96 ± 236.577 ops/sec
```

2. Message sizes - iroh version

```
Workload 1:
  Run 0: 25 store_block_e operations
    Min: 536, Max: 665, Avg: 659.5
  Run 1: 25 store_block_e operations
    Min: 536, Max: 665, Avg: 659.5
  Run 2: 25 store_block_e operations
    Min: 536, Max: 665, Avg: 659.5
  Run 3: 25 store_block_e operations
    Min: 536, Max: 665, Avg: 659.5
  Run 4: 25 store_block_e operations
    Min: 536, Max: 665, Avg: 659.5

Workload 2:
  Run 0: 49 store_block_e operations
    Min: 536, Max: 665, Avg: 662.2
  Run 1: 49 store_block_e operations
    Min: 536, Max: 665, Avg: 662.2
  Run 2: 49 store_block_e operations
    Min: 536, Max: 665, Avg: 662.2
  Run 3: 49 store_block_e operations
    Min: 536, Max: 665, Avg: 662.2
  Run 4: 49 store_block_e operations
    Min: 536, Max: 665, Avg: 662.2

Workload 4:
  Run 0: 97 store_block_e operations
    Min: 536, Max: 665, Avg: 663.6
  Run 1: 67 store_block_e operations
    Min: 536, Max: 665, Avg: 663.0
  Run 2: 97 store_block_e operations
    Min: 536, Max: 665, Avg: 663.6
  Run 3: 97 store_block_e operations
    Min: 536, Max: 665, Avg: 663.6
  Run 4: 97 store_block_e operations
    Min: 536, Max: 665, Avg: 663.6

Workload 8:
  Run 0: 192 store_block_e operations
    Min: 536, Max: 666, Avg: 664.8
  Run 1: 191 store_block_e operations
    Min: 536, Max: 666, Avg: 664.8
```

```
  Run 2: 191 store_block_e operations
    Min: 536, Max: 666, Avg: 664.8
  Run 3: 190 store_block_e operations
    Min: 536, Max: 666, Avg: 664.8
  Run 4: 191 store_block_e operations
    Min: 536, Max: 666, Avg: 664.8

Workload 16:
  Run 0: 374 store_block_e operations
    Min: 536, Max: 666, Avg: 665.4
  Run 1: 373 store_block_e operations
    Min: 536, Max: 666, Avg: 665.4
  Run 2: 374 store_block_e operations
    Min: 536, Max: 666, Avg: 665.4
  Run 3: 374 store_block_e operations
    Min: 536, Max: 666, Avg: 665.4
  Run 4: 373 store_block_e operations
    Min: 536, Max: 666, Avg: 665.4

Workload 32:
  Run 0: 720 store_block_e operations
    Min: 536, Max: 666, Avg: 665.7
  Run 1: 722 store_block_e operations
    Min: 536, Max: 666, Avg: 665.7
  Run 2: 720 store_block_e operations
    Min: 536, Max: 666, Avg: 665.7
  Run 3: 723 store_block_e operations
    Min: 536, Max: 666, Avg: 665.6
  Run 4: 722 store_block_e operations
    Min: 536, Max: 666, Avg: 665.6

Workload 64:
  Run 0: 877 store_block_e operations
    Min: 536, Max: 666, Avg: 665.7
  Run 1: 396 store_block_e operations
    Min: 536, Max: 666, Avg: 665.4
  Run 2: 1373 store_block_e operations
    Min: 536, Max: 667, Avg: 666.1
  Run 3: 3 store_block_e operations
    Min: 536, Max: 664, Avg: 600.0
  Run 4: 2 store_block_e operations
    Min: 536, Max: 664, Avg: 600.0

Workload 128:
  Run 0: 4 store_block_e operations
    Min: 536, Max: 664, Avg: 616.0
  Run 1: 2544 store_block_e operations
    Min: 536, Max: 667, Avg: 666.5
```

```
  Run 2: 2546 store_block_e operations
    Min: 536, Max: 667, Avg: 666.5
  Run 3: 2535 store_block_e operations
    Min: 536, Max: 667, Avg: 666.3
  Run 4: No store_block_e operations found

Workload 256:
  Run 0: 1697 store_block_e operations
    Min: 536, Max: 667, Avg: 665.9
  Run 1: 4547 store_block_e operations
    Min: 536, Max: 667, Avg: 666.7
  Run 2: 5 store_block_e operations
    Min: 536, Max: 664, Avg: 600.0
  Run 3: 6 store_block_e operations
    Min: 536, Max: 664, Avg: 632.0
  Run 4: 5 store_block_e operations
    Min: 536, Max: 664, Avg: 625.6

Workload 512:
  Run 0: 8 store_block_e operations
    Min: 536, Max: 600, Avg: 592.0
  Run 1: 7645 store_block_e operations
    Min: 536, Max: 667, Avg: 666.8
  Run 2: 7741 store_block_e operations
    Min: 536, Max: 667, Avg: 666.8
  Run 3: 10 store_block_e operations
    Min: 536, Max: 665, Avg: 632.1
  Run 4: 8 store_block_e operations
    Min: 536, Max: 600, Avg: 592.0

Workload 1024:
  Run 0: 3 store_block_e operations
    Min: 536, Max: 600, Avg: 578.7
  Run 1: 12339 store_block_e operations
    Min: 536, Max: 669, Avg: 667.2
  Run 2: 12 store_block_e operations
    Min: 536, Max: 665, Avg: 621.6
  Run 3: 12280 store_block_e operations
    Min: 536, Max: 669, Avg: 667.3
  Run 4: 12367 store_block_e operations
    Min: 536, Max: 669, Avg: 667.2
```

3. Confidence intervals for throughputs - UDP version

```
Throughput Summary with 95% CI:
Workload 1:
  Send: 1.01 ± 0.002 ops/sec
  Store: 1.01 ± 0.002 ops/sec
```

```
Workload 2:
  Send: 1.97 ± 0.002 ops/sec
  Store: 1.97 ± 0.002 ops/sec
Workload 4:
  Send: 3.89 ± 0.003 ops/sec
  Store: 3.89 ± 0.003 ops/sec
Workload 8:
  Send: 7.66 ± 0.008 ops/sec
  Store: 7.66 ± 0.008 ops/sec
Workload 16:
  Send: 15.03 ± 0.023 ops/sec
  Store: 15.03 ± 0.023 ops/sec
Workload 32:
  Send: 29.08 ± 0.020 ops/sec
  Store: 29.08 ± 0.020 ops/sec
Workload 64:
  Send: 55.63 ± 0.121 ops/sec
  Store: 55.63 ± 0.121 ops/sec
Workload 128:
  Send: 102.40 ± 0.737 ops/sec
  Store: 102.38 ± 0.752 ops/sec
Workload 256:
  Send: 190.57 ± 0.754 ops/sec
  Store: 190.53 ± 0.754 ops/sec
Workload 512:
  Send: 305.32 ± 3.895 ops/sec
  Store: 305.26 ± 3.909 ops/sec
Workload 1024:
  Send: 508.99 ± 8.156 ops/sec
  Store: 508.27 ± 7.283 ops/sec
```

4. Message sizes - UDP version

```
Workload 1:
  Run 0: 25 store_block_e operations
    Min: 536, Max: 665, Avg: 659.5
  Run 1: 25 store_block_e operations
    Min: 665, Max: 665, Avg: 665.0
  Run 2: 25 store_block_e operations
    Min: 665, Max: 665, Avg: 665.0
  Run 3: 25 store_block_e operations
    Min: 665, Max: 666, Avg: 665.0
  Run 4: 25 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0

Workload 2:
  Run 0: 49 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
```

```
  Run 1: 49 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 2: 49 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 3: 49 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 4: 49 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0

Workload 4:
  Run 0: 97 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 1: 97 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 2: 97 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 3: 97 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0
  Run 4: 97 store_block_e operations
    Min: 666, Max: 666, Avg: 666.0

Workload 8:
  Run 0: 191 store_block_e operations
    Min: 666, Max: 667, Avg: 666.2
  Run 1: 191 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 2: 191 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 3: 191 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 4: 191 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0

Workload 16:
  Run 0: 375 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 1: 375 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 2: 376 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 3: 376 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 4: 376 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0

Workload 32:
  Run 0: 727 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
```

```
  Run 1: 727 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 2: 727 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 3: 726 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 4: 728 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0

Workload 64:
  Run 0: 1394 store_block_e operations
    Min: 667, Max: 667, Avg: 667.0
  Run 1: 1385 store_block_e operations
    Min: 667, Max: 668, Avg: 667.1
  Run 2: 1392 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 3: 1393 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 4: 1391 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0

Workload 128:
  Run 0: 2537 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 1: 2547 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 2: 2558 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 3: 2563 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 4: 2594 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0

Workload 256:
  Run 0: 4727 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 1: 4784 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 2: 4773 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 3: 4769 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 4: 4764 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0

Workload 512:
  Run 0: 7830 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
```

```
  Run 1: 7571 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 2: 7603 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 3: 7573 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0
  Run 4: 7581 store_block_e operations
    Min: 668, Max: 668, Avg: 668.0

Workload 1024:
  Run 0: 12474 store_block_e operations
    Min: 668, Max: 670, Avg: 668.3
  Run 1: 12800 store_block_e operations
    Min: 669, Max: 670, Avg: 669.2
  Run 2: 12540 store_block_e operations
    Min: 669, Max: 670, Avg: 669.2
  Run 3: 12992 store_block_e operations
    Min: 669, Max: 670, Avg: 669.2
  Run 4: 12729 store_block_e operations
    Min: 669, Max: 670, Avg: 669.2
```

C Smartphone Agent Specification

Model: Motorola Moto G04s

Operating System: Android 14

Processor: Unisoc T606 (Octa-core: 2xCortex-A75 1.6 GHz, 6x Cortex-A55 1.6 GHz)

RAM: 4 GB

Internal Storage: 64 GB (expandable via microSD)

Battery: 5000 mAh with 15 W fast charging

Connectivity: Dual SIM, 4G LTE, Wi-Fi 802.11 a/b/g/n/ac, Bluetooth 5.0