Gossamer: A novel programming paradigm for rack-wide applications

by

Noaman Ahmad
BS CS, Lahore University of Management Sciences, 2018

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2025

Chicago, Illinois

A brief dedication to someone you care about. For example, “Dedicated to my cats, Neo and Trinity, who are purrfect in every way.”.

An example of “Dedication” can be found on page 15 of the thesis manual[1].

[1]`http://grad.uic.edu/sites/default/files/pdfs/ThesisManual_rev_06Oct2016.pdf`

ACKNOWLEDGMENT

A page or two so of shout-outs to people you appreciate. Don’t forget your advisor and committee members!

NA

CONTRIBUTIONS OF AUTHORS

This section should give a rough overview of each chapter in the thesis, highlighting your contributions. Most importantly, for each one of your papers you are quoting, this section should briefly describe what each author's role / contribution was.

An example of "Contribution of Authors" section is on page 3 of the University's guide to iThenticate[1].

[1] `http://grad.uic.edu/sites/default/files/pdfs/Introduction_to_Screening_Your_Thesis_or_Dissertation_using_iThenticate-final_a.pdf`

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

SUMMARY

One to two page summary of the entire work. Like a long abstract.

CHAPTER 1

INTRODUCTION

Safe access to shared objects is fundamental to many multi-threaded programs. Conventionally, this is achieved through *locking*, or in some cases through carefully designed lock-free data structures, both of which are implemented using atomic compare-and-swap (CAS) operations. By their nature, atomic instructions do not *scale* well: atomic instructions must not be reordered with other instructions, often starving part of today's highly parallel CPU pipelines of work until the instruction has retired. This effect is exacerbated when multiple cores are accessing the same object, resulting in the combined effect of frequent cache misses and cores waiting for each other to release the cache line in question, while the atomic instructions prevent them from doing other work.

Delegation (1; 2; 3; 4; 5; 6; 7; 8; 9; 10), also known as message-passing or light-weight remote procedure calls (LRPC), offers a highly scalable alternative to locking. Here, each shared object[1] is placed in the care of a single core (*trustee* below). Using a shared-memory message passing protocol, other cores (clients) issue requests to the trustee, specifying operations to be performed on the object.

Compared to locking, where threads typically contend for access, and may even suspend execution to wait for access, delegation requests from different clients are submitted to the

[1]Here, we use *object* to mean a data structure that would be protected by a single lock.

trustee in parallel and without contention. This dramatically reduces the cost of coordination for congested objects. The operations/critical sections are applied sequentially in both designs: by each thread using locks, or by the trustee using delegation; here delegation may benefit from improved locality at the trustee. Together, this translates to much higher maximum per-object throughput with delegation vs. locking.

However, under medium or low contention, classical delegation struggles to compete with locking: the latency and overhead of request transmission, request processing and response transmission are insignificant compared to the cost of contending for a lock, but can be substantial compared to the cost of acquiring an *uncontended* lock.

The main contributions of this dissertation are summarized below:

- Trust<T>: a model for efficient, multi-threaded, delegation-based programming with shared objects leveraging the Rust type system.
- Gossamer: an extension of Trust<T> that enables a normal delegation based application to run on and utilize the resources of a rack with minimal changes to application code.

The rest of this chapter introduces both of these and then outlines the rest of this dissertation.

1.1 ***Trust<T>***

We present *Trust<T>* (pronounced *trust-tee*), a programming abstraction and runtime system which provides safe, high-performance access to a shared object (or property) of type T.Briefly, a *Trust<T>* provides a family of functions of the form:

$$\mathsf{apply}(\mathsf{c} : \mathsf{FnOnce}(\&\mathsf{mut}\ \mathsf{T}) \to \mathsf{U}) \to \mathsf{U},$$

which causes the closure c to be safely applied to the property (of type T), and returns the return value (of type U) of the closure to the caller. Here, FnOnce denotes a category of Rust closure types, and &mut denotes a mutable reference. (A matching set of non-blocking functions is also provided, which instead executes a callback closure with the return value.)

Critically, access to the property is only available through the *Trust<T>* API, which taken together with the Rust ownership model and borrow checker eliminates any potential for race conditions, given a correct implementation of *apply*. Our implementation of *Trust<T>* uses pure delegation. However, the design of the API also permits lock-based implementations, as well as hybrids.

Beyond the API, *Trust<T>* provides a runtime for scheduling request transmission and processing, as well as lightweight user threads (fibers below). This allows each OS thread to serve both as a Trustee, processing incoming requests, and a client. Multiple outstanding requests can be issued either by concurrent synchronous fibers or an asynchronous programming style.

1.2 Gossamer

This work extends *Trust*<*T*> to build a rack-wide programming model. *Gossamer* aims to provide a high performance and easy to program in, framework that can take advantage of the increased resources available in a rack. Using the API provided by *Gossamer* programmers will be able to code their applications like a normal delegation application with very few changes. Most of the distribution of work is handled behind the scenes, but programmers have the option to customize where the worker threads should run and where data should be held in the rack if they so choose.

One inherent problem that restricts performance in distributed settings is the higher latency caused by network traversal. *Gossamer* overcomes that by using RDMA along with taking advantage of fibers. RDMA provides sub-micro second one way latency that is comparable to that of permanent storage on single machine. While RDMA solves the problem with high latency, it does not scale well with increasing number of connections per machine. To solve this *Gossamer* establishes only one connection per hardware thread for each remote machine and uses many fibers to utilize the available throughput to full extent.

Gossamer implements delegation in a way that applications will have the illusion of running on the same machine. This can be done by mapping the server memory to a client's address space using RDMA. This means that various things that are normally only possible in shared memory can be done in a distributed setting. Some examples include spawning a fiber, joining a fiber and accessing a shared object.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces *Trust*<*T*>, a delegation based programming abstraction that provides safe access to shared data. Chapter 3 presentes *Gossamer*, an extension of *Trust*<*T*>, that allows single-machine, multi-threaded applications built using *Trust*<*T*>, to run on a rack consisting of many machines with minimal changes. Chapter 4 discusses future work and provides a conclusion.

CHAPTER 2

***Trust*< *T*>**

CHAPTER 3

GOSSAMER

3.1 Background on RDMA

RDMA is a networking approach that allows one machine to directly access a remote machine's memory. It uses the user-level zero-copy transfers to minimize the involvement of remote machine's operating system and CPU as opposed to traditional TCP/IP stack that involve both on each machine heavily. There are many implementations of this concept (11; 12; 13; 14), most popular of which are InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (internet Wide Area RDMA Protocol). The results presented in this paper are obtained by using RoCE.

3.1.1 RDMA API

RDMA hosts communicate using queue pairs (QPs) that consist of a send queue and a receive queue, and are maintained by the NIC. Applications post operations to these QPs by using functions called *verbs.* For remote access the remote machine first needs to register a memory region with the NIC. The NIC driver pins this region in physical memory. The address and a key related to this region then needs to be exchanged between the machines out of band (i.e. without using RDMA). After this exchange, the remote memory can be accessed without involving either of remote operating system or CPU. This is called *RDMA Memory Semantics*, and uses *verbs READ* and *WRITE.*

RDMA also provides *Messaging Semantics* that use *verbs SEND* and *RECV*. In this case receiver has to post a *RECV verb* before the sender can send the data. In this regard it is similar to an unbuffered sockets implementation. Just like *Memory Semantics*, *Messaging Semantics* also bypass the remote kernel but unlike *Memory Semantics*, it has to involve remote CPU to post a *RECV*. These *verbs* also have slightly lower latency than *READ* and *WRITE* (15; 16).

3.1.2 Transport types

RDMA transports are either connected or unconnected (also called datagram), and either reliable or unreliable. Connected transport require a one-to-one connection between two QPs. If an application wants to communicate with N machines, it will need to create N QPs. With unconnected transport one QP can communicate with many QPs. For reliable transport NIC uses acknowledgments to guarantee in-order delivery and return an error code on failure, while unreliable transport does not provide any such guarantee. InfiniBand and RoCE use lossless link layer, so even in case of unreliable transports, losses are pretty rare and happen because of bit error or link failure. In case of connected transport a failure will break the connection. Not all transports provide all of the verbs. Table Table I gives an overview of transport types and the verbs they support. Current implementations of RDMA only provide reliable connected (RC), unreliable connected (UC) and unreliable datagram.

3.2 Gossamer Design

Gossamer allows developers to write rack-wide applications just like single machine applications. Programmers will use the delegation framework to write applications that would be distributed by gossamer with very few changes to the code. Figure 1 shows a high level view

Verb	RC	UC	UD
SEND/RECV	✓	✓	✓
WRITE	✓	✓	✗
READ	✓	✗	✗

TABLE I: Operations supported by each transport type.

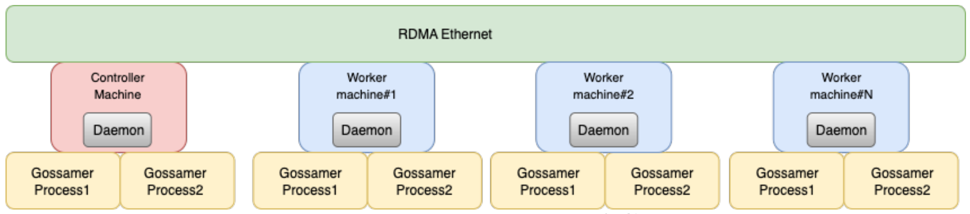

Figure 1: High-level design of Gossamer

of what a rack with multiple *Gossamer* applications would look like. Developers would use the controller machine to interact with the rack and launch applications. Here controller machine is just a normal machine running *Gossamer* processes like any other machine in the rack, the only significance is that this will be the machine that users log into for terminal access. Each machine in the rack would have a daemon running that would always be listening for instructions from the controller machine. The daemon is responsible for determining the topology of the rack and managing the applications. A machine can have multiple applications running on it at the same time. Rack applications that are running simultaneously are isolated from each other like multiple processes on a single machine. The rest of this section describes some design details that are particularly interesting.

3.2.1 Gossamer Daemon

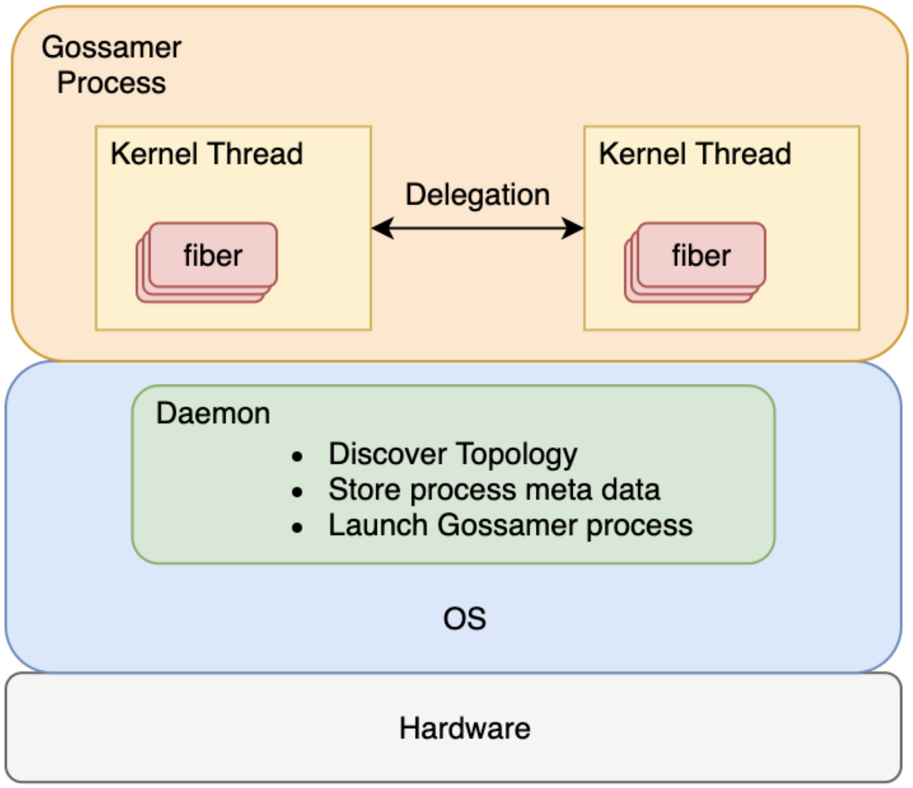

Figure 2: A Gossamer process on one machine

As shown in Figure 2, each machine has *Gossamer* daemon running. The daemon is responsible for managing any *Gossamer* processes running on the rack at any time. To launch an application, users will connect with the controller machine and tell the daemon to start the process specifying how many kernel threads the process should use. The daemon will contact the remote machines and tell the daemons to launch the process. The daemon will store metadata like gossamer-id, machine-ip etc. The daemon is also responsible for detecting crashes on any local instance of a *Gossamer* process and terminating across the whole rack. This means that *Gossamer* processes are fate sharing in the same way single machine processes crash if a single thread crashes.

3.2.2 Gossamer Process

To launch a *Gossamer* process, the user logs into the controller machine and starts the application there. Since parts of *Gossamer* rely on similar memory layout (as discussed in 3.2.8), each machine in the rack should be populated with the same binary ahead of time. The application needs to have its *main* function call a helper function from the *Gossamer* library and pass a function that would act as the real *main* function along with how many threads the application needs to use across the whole rack. This function will be referred to as *gsm_main* for the rest of the paper. The process consists of many fibers per kernel thread for concurrent operation as shown in Figure 2. Fibers are lightweight, userspace threads that can spawn on any kernel thread that is part of the *Gossamer* process, and on any machine in the rack. Fibers that need to access shared data make delegation requests consisting of closures that modify the data as needed. From the point of view of th operating system, a single instance of *Gossamer* process behaves like any other normal process. The main difference would be that instead of using system calls like *getpid()*, a *Gossamer* process contacts the daemon to get any metadata needed.

3.2.3 Trustee, Trust and Property

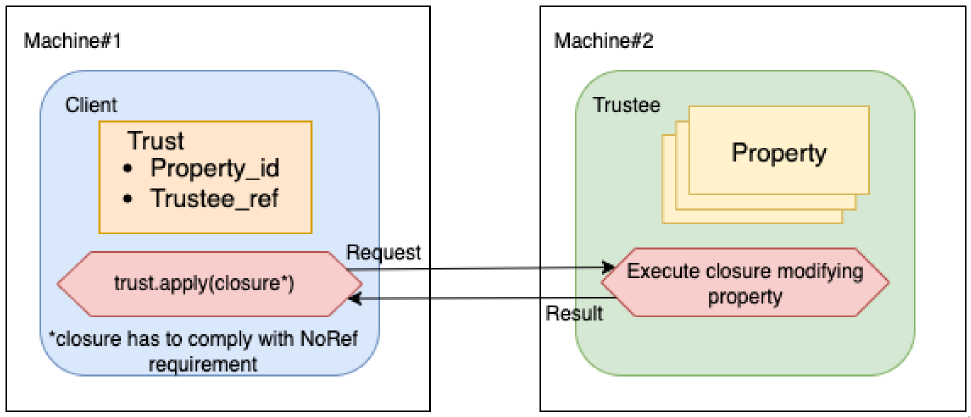

Figure 3: Trustee is a worker and clients can delegate work using a trust that holds information about property and trustee

Gossamer builds on the Trust/Trustee concept from (17), which we introduce briefly here. A *Trustee* is a worker fiber that processes delegation requests and sends back the response. The application can entrust some data to a trustee if the delegated work needs access to it and receive a *Trust* that holds relevant metadata. This entrusted data is referred to as *Property*. The Trust can be used to delegate any work that needs access to this property. Figure 3 shows the relationship between Trust, Trustee and Property. A Trust can be cloned and shared with other fibers so that they can also start delegating work that needs access to same property. Multiple properties can be entrusted to a single trustee at the same time. Trustees can act as remote or local trustees based on where the fiber trying to access the property lives in the rack.

```
let property = HashMap::new();
let trust = gossamer::entrust(property);
gossamer::spawn(|| {
    trust.apply(|property| property.insert(``First Greeting'',``Hello''));
    trust.apply_then(|property| property.insert(``Second Greeting'',``Hi''),
        |_| println!(``added second greeting''));
});
```

TABLE II: A small example of delegation code

3.2.4 Delegation Workflow

Gossamer provides both blocking and nonblocking/asynchronous delegation operations. In case of blocking operations, the delegating fiber will wait for the response from the trustee before continuing. For nonblocking delegation, the delegating fiber will instead provide a callback closure to be executed upon completion of the request, and continue without waiting. First we will describe the workflow for blocking delegation requests and then specify how that differs from the nonblocking delegation. Each fiber issues a request that is put in a pending queue before voluntarily yeilding the runtime. Each thread has a fiber that periodicaly checks for any incoming responses and sends any requests in the pending queue. As there will be many fibers running on each thread this will allow the polling fiber to send multiple requests as a larger batch, increasing the throughput of the system. On the trustee side, one of the fibers is dedicated to polling for any incoming requests. Since any incoming requests from the same thread will be contigous in memory, it can complete all of them and then send all of the responses in a single batch. The fiber that polls for reponses on the requesting thread will then process the responses and add the corresponding fibers back in to the ready to run queue. The main difference for async delegation is that instead of yeilding after enqueuing the request, it just keeps going and enqueues any subsequent requests as well. This fiber will yield the runtime when the pending queue is full, after which the response polling fiber can run and send all of the requests to the respective trustees. Once it receives the responses, it can execute any callback closures the user might have provided. This means that there is no benefit to having multiple fibers per thread that issue requests as that is not needed for batching and the fibers will be yielding far less frequently. Table II shows a small example program that uses both types of delegation.

3.2.5 NoRef

Gossamer leverages the Rust type system to prevent the sending of references over delegation channels. One of the first challenges that arise when extending delegation to multiple machines is the fact that any pointers or references captured by a delegated closure will not be valid on a remote machine. This reference or pointer will result in undefined behavior on any machine other than where it originated. To prevent this, *Gossamer* uses a combination of rust features named *auto_traits* and *negative_impls*. In rust, traits define the behavior of data types. They inform the compiler what functionality a type has and can be used to restrict which data types can be passed to a generic function. Auto_traits is a feature that automatically implements a trait for every data type, be it an existing type or user defined. A negative implementation is used to exclude a data type from the auto implementation. *Gossamer* defines an auto trait called *NoRef*. Then all the types that contain a reference or raw pointer are given negative implementation. The function *apply*, that is used to send delegation messages as described in previous section, requires all the closures to comply with the NoRef trait as shown in Figure 3. This however limits severly what type of data can be captured by any delegated closure as many frequently used types like strings and vectors contain pointers internally. To get around that *Gossamer* prvides a way to send any data type that can be serialized along with the closure as an extra parameter to the delegated closure. If the programmer makes a mistake and tries to use a closure that has captured a reference for delegation, a custom error message at compile time will inform them what the issue is and direct them to use the serialization method instead.

3.2.6 Reference counting for Trusts

Trusts in *Gossamer* act as rust smart pointers that own the property, since they can be used to access and modify the property behind them. This means that we need to make sure that when all of the trusts are dropped, the property is also dropped and the associated memory is freed so as not to have memory leakage. To acheive that trusts need to be reference counted, but a naive integer count will not suffice due to a combination of the following two issues.

- *Gossamer* does not support a blocking delegation operation when in the middle of another delegation request. For example calling *apply* on a trust from within a closure that itself is used for delegation will cause the system to hang and never finish.
- If the integer used for counting references, let's call it refcount, is incremented and decremented asynchronously, i.e. with a nonblocking delegation call, it could lead to use after free bugs. Let's consider the following scenario: Thread A clones a trust with only one reference and sends an async request to increment its refcount. The cloned trust is then

sent to Thread B that drops it, sending another async request to decrement its refcount. While the requests issued by the same fiber are guaranteed to be completed in the same order they are issued, there is no such guaranty across multiple threads or even fibers. It is entirely possible that the decrement request is processed first, making the refcount zero, which results in the property being dropped. Thread A however still has a valid trust to this property which it can use, expecting the property to still be available.

Not being able to use async delegation to increment and decrement the refcount leads to not being able to clone a trust from within a delegated context, which can quite restrictive. To get around that, *Gossamer* uses a new way to keep track of how many trusts are active at any time. Each trust is given a unique id at the time of creation, be it a new trust or a clone of an existing one. Instead of just using an integer, *Gossamer* uses a set of these ids associated with each property. Instead of incrementing or decrementing the refcount, clone and drop both issue a delegation request involving a symmetric difference operation. Symmetric difference is defined as adding an element to a set if it is not already a member, and removing it from the set if it is. This way, regardless of the order in which requests originating in clone and drop are processed, the first one will add the trust id to the set and later one will remove it, allowing us to use async delegation for both. This, in turn, enables us to clone a trust within a delegated context.

3.2.7 Request size and TCP fallback

As describe in 3.2.4, the requsts going from a client thread to the same trustee will be batched and on trustee side the requests coming from a single thread will all be contigous in memory. This is achieved by the trustee having a pre-allocated sapce in memory (called requestSlot) for incoming requests from each client thread. This limits how many requests a client thread can send in a single batch depending on how much data is being sent with each request. As an example if each request request captures 32 bytes of captured data, the total request size is 40 bytes. Next section goes into detail about the format of a *Gossamer* delegation request. Assuming 1kB requestSlot, one batch can have at most 25 requests. While this allows us to optimize for small requests it also places a hard limit on how much data a request can capture i.e. the size of the requestSlot. To work aound this limit and support larger requests, *Gossamer* uses TCP to send any captured data larger than that separately while the request header goes in the requestSlot as normal. This degrades the performance severly as TCP is much slower than RDMA. We use TCP here instead of RDMA because, as mentioned in 3.1.1, RDMA needs the memory used to be pre-registered with the NIC. This would also place an upper limit on how much data can be sent using RDMA at once. Since there would be an upper limit anyway and larger requests are not the focus for *Gossamer*, we decided to opt for the simpler design in place at the moment.

3.2.8 Request Format

As discussed in the previos section, the number of requests sent in batch depends on the size of the requests plus the size of any headers sent with the request. The necessary parts to process a *Gossamer* request are as follows:

- The closure. Closures in rust are fat pointers consisting of a vtable that points to where the code is in the text section along with the size of captured data, and a data pointer that points to the captured data.
- The pointer to where the property is located on trustee.
- Length of serialized data.
- The captured data.
- Any serialized data.

While we can't avoid sending the captured data, serialized data and the propoerty pointer, *Gossamer* employs some strategies to minimize the information that needs to be sent inside the request slot. As the data pointer within the closure is not going to be valid on a remote machine, there is no need to include that in the request. Since the same binary is running on all the machines in the system, they all have access to any information known at compile time. This includes all of the information in the vtable, provided the vtable pointer. Here, we make the observation that for the vast majority of the runtime of an application, most of the requests in the request slot will be about a single closure or a few closures at most, meaning we can have a small lookup table for a few vtable pointers in the request slot instead of including one with every request. Combining that with the start point of data received, the trustee can reconstitute the closure locally, removing the need to send the closure in the request slot. If there are indeed more closures in the request slot than will fit in the lookup table, any extras can be sent in the request slot. This will reduce the data being sent in the common case with minimal overhead, but will be no more expansive in the special case. Similarly if the serialized data has a size known at compile time the trustee already has access to it, so the only time it needs to be sent with the request is if it is not known at compile time. One thing to note here is that we rely on same vtable pointer to be the valid on all of the machines which is not the case normally due to linux address space layout randomization (ASLR), making disabling it a necessity for *Gossamer* to function.

3.3 Plots

Various plot comparing gossamer with eRPC and graph500 reference code.

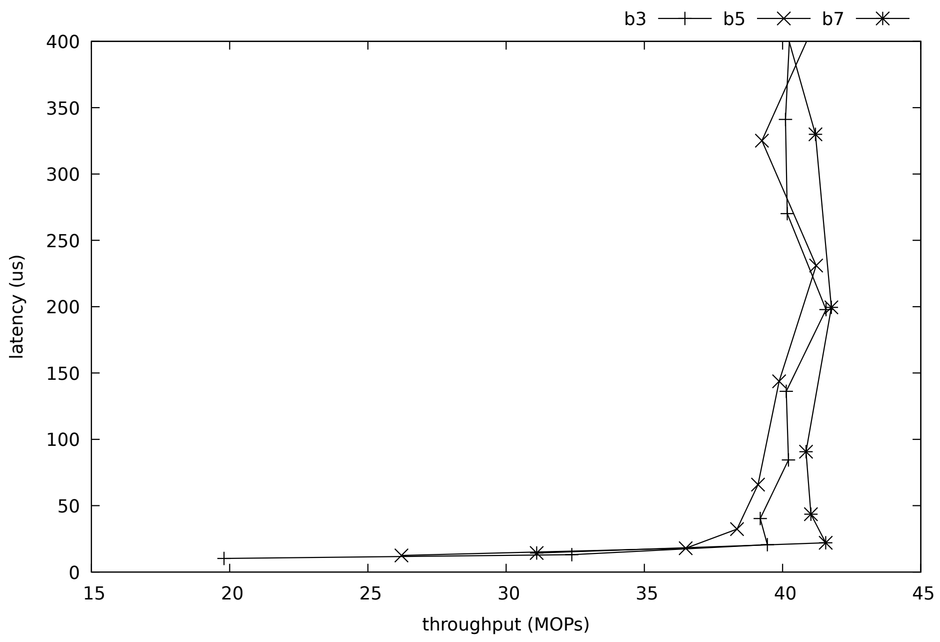

Figure 4: throughput vs latency for different batch sizes for eRPC.

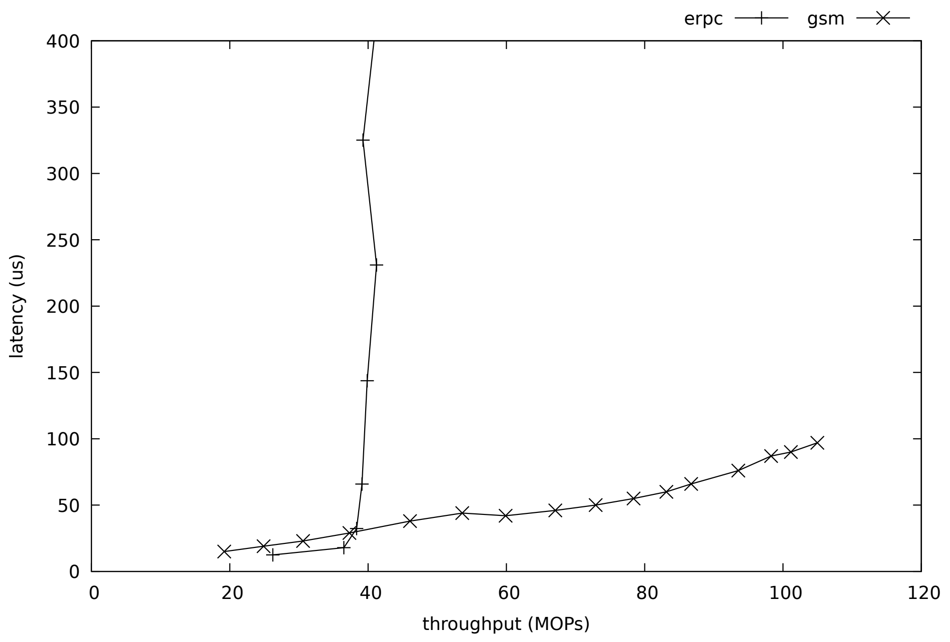

Figure 5: throughput vs latency under load for gsm and eRPC with two machines and 14 threads per machine.

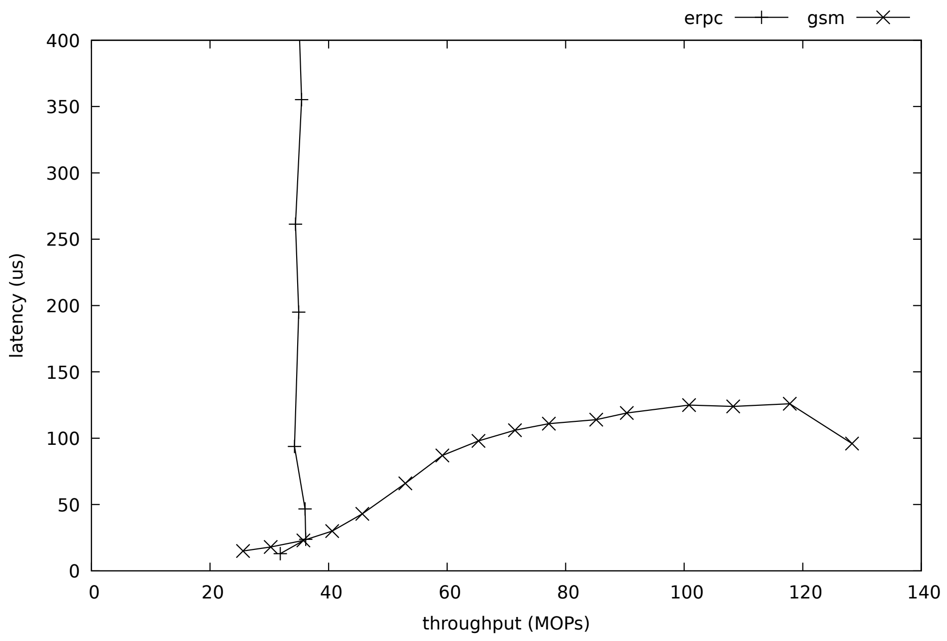

Figure 6: throughput vs latency under load for gsm and eRPC with two machines and 28 threads per machine.

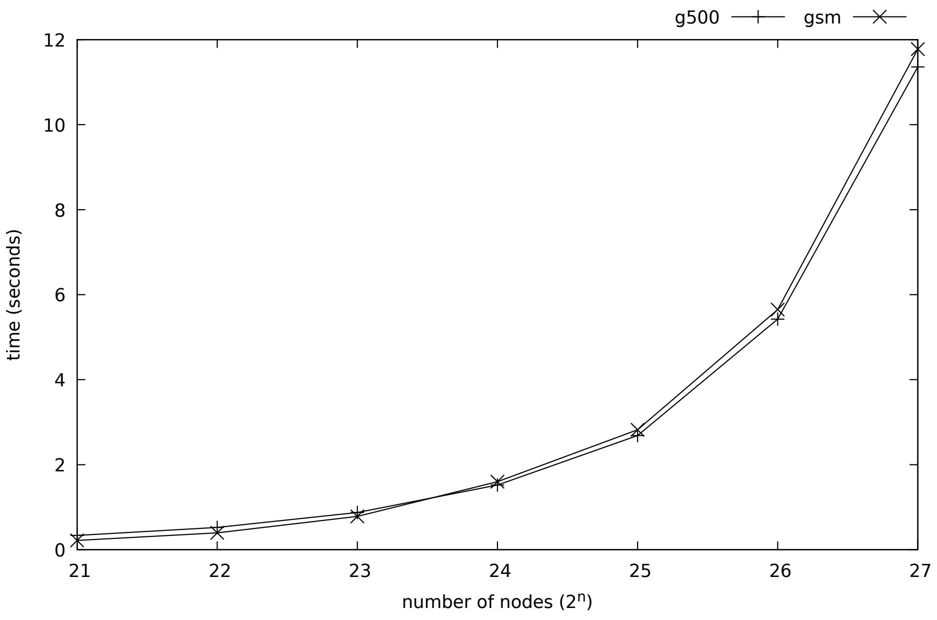

Figure 7: Time taken for compute shortest path on a graph with 2^{26} nodes and an average degree of 32. Each successive point repesents an additional machine from 16 Cloudlab xl170 nodes with a total of 160 threads.

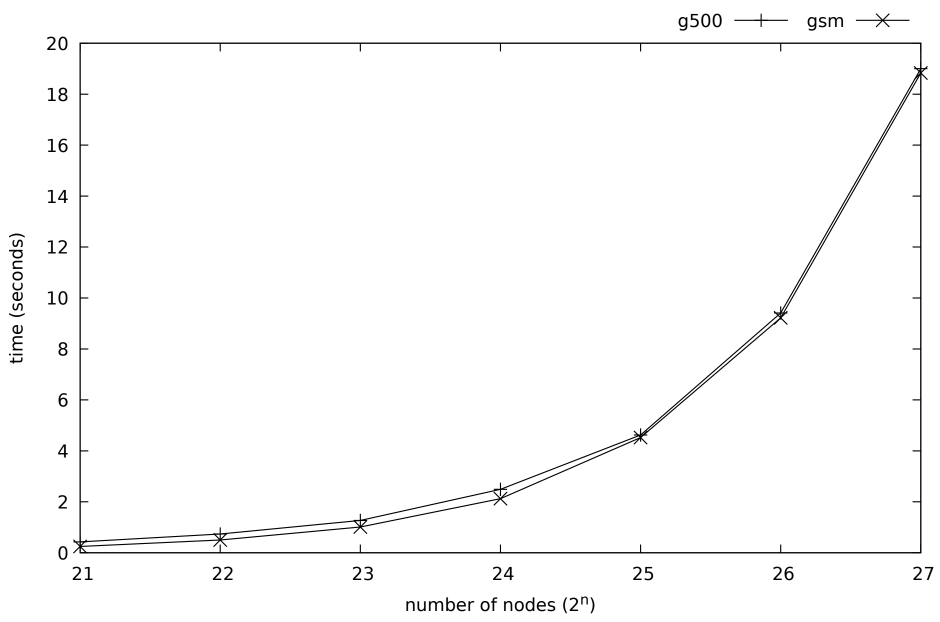

Figure 8: Time taken for compute shortest path on graphs with 2^n nodes and an average degree of 32. 4 machines with a total of 112 threads.

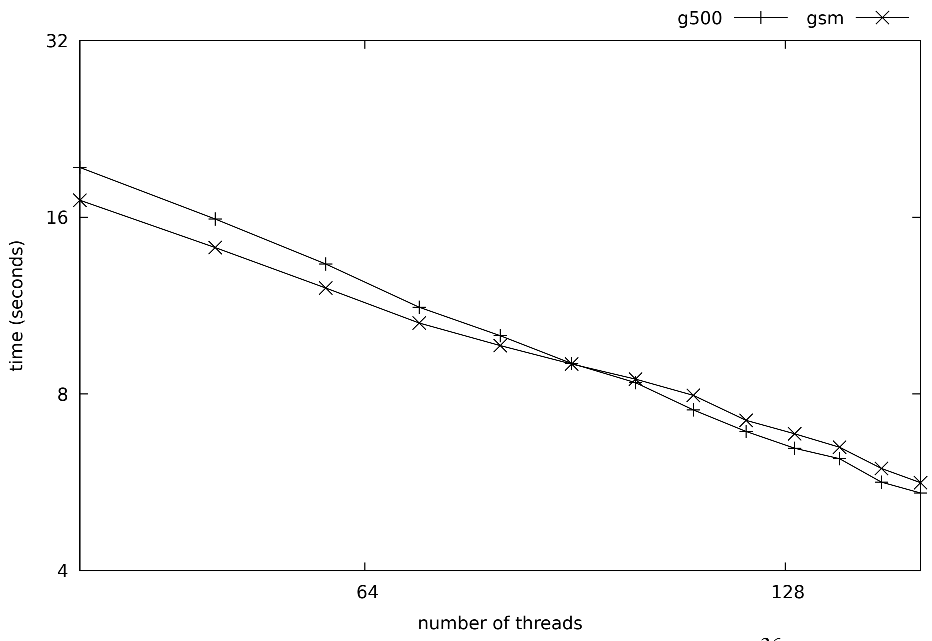

Figure 9: Time taken for compute shortest path on a graph with 2^{26} nodes and an average degree of 32. 16 Cloudlab xl170 nodes with a total of 160 threads.

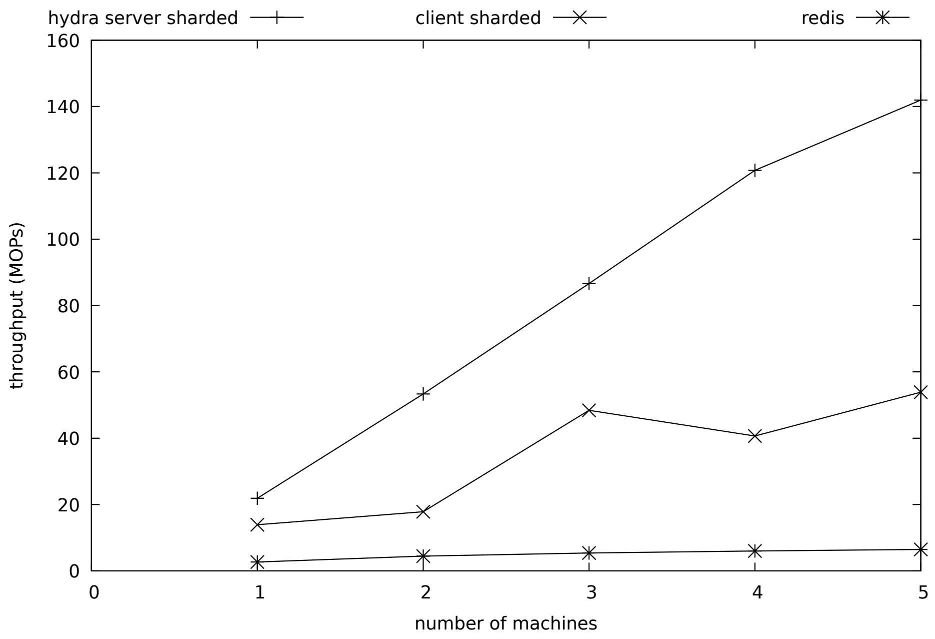

Figure 10: Throughput of hydra vs redis. Each server machine has 6 trustees in case of hydra and 6 intances of redis in case of redis store.

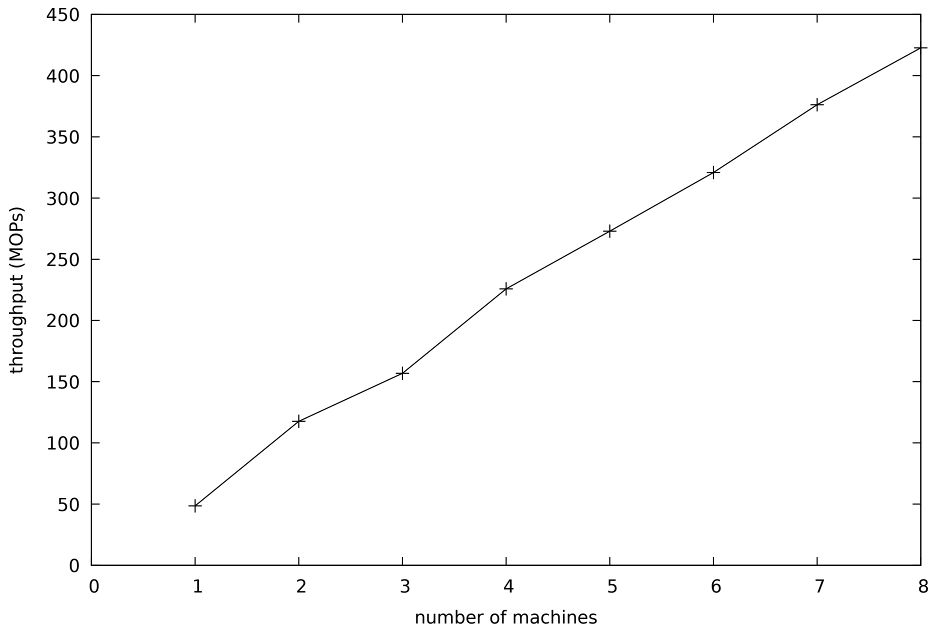

Figure 11: Throughput of the key-value store. The number of servers is varied here with each one having 6 dedicated trustee and 26 threads to handle incoming client queries. The clients are 4 machines with 768 threads over 64 cores. This experiment was conducted on cloudlab r6515 nodes.

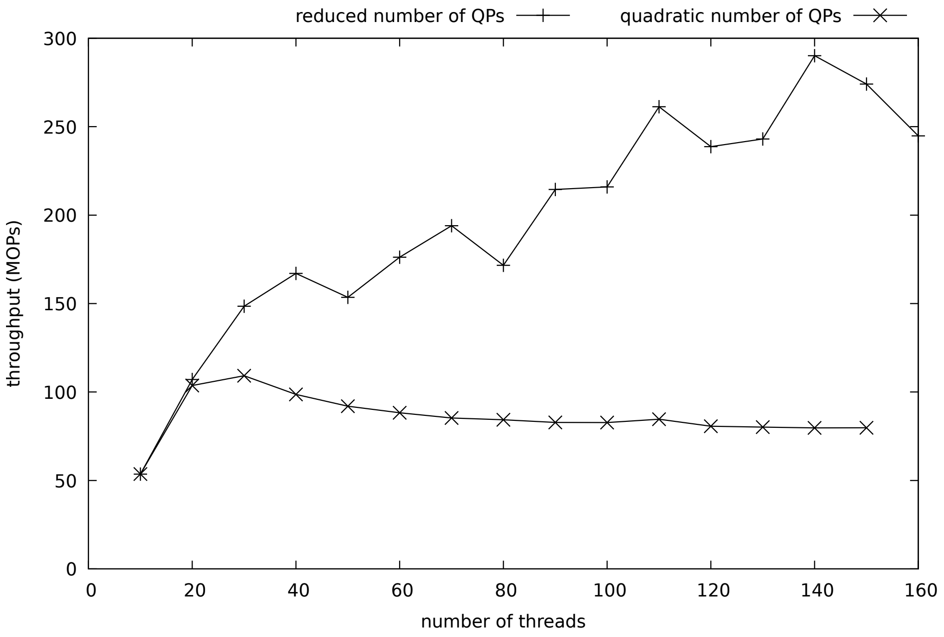

Figure 12: Throughput of RDMA writes in MOPs for quadratic and reduced number of QPs. Each write is 32 Bytes. and th experiment was conducted on 5 r6515 machines on cloudlab.

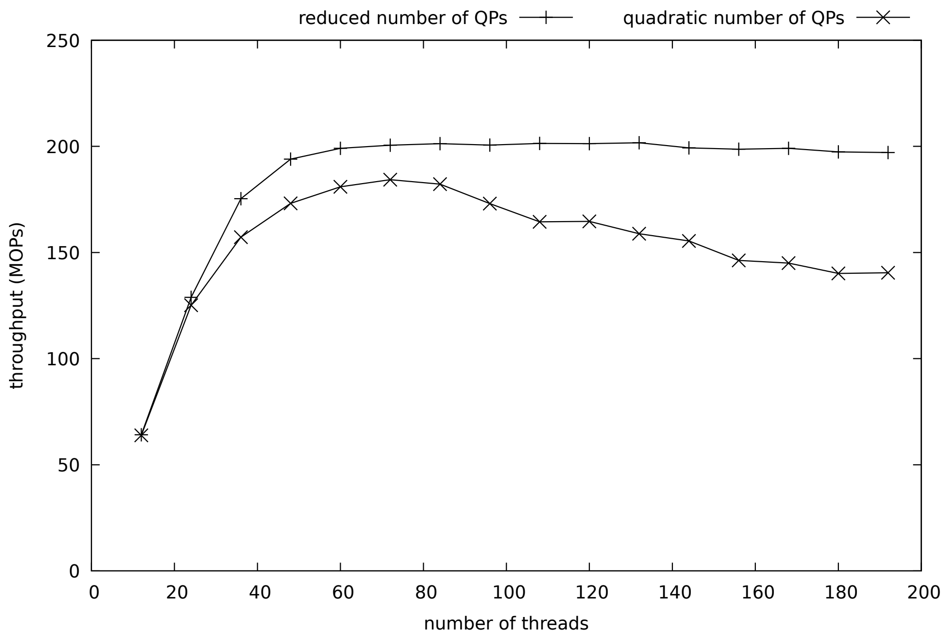

Figure 13: Throughput of RDMA writes in MOPs for quadratic and reduced number of QPs. Each write is 256 Bytes. and th experiment was conducted on 6 r6515 machines on cloudlab.

CHAPTER 4

CONCLUSION

CITED LITERATURE

1. Dice, D., Marathe, V. J., and Shavit, N.: Flat-combining numa locks. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pages 65–74. ACM, 2011.

2. Calciu, I., Dice, D., Harris, T., Herlihy, M., Kogan, A., Marathe, V., and Moir, M.: Message passing or shared memory: Evaluating the delegation abstraction for multicores. In International Conference on Principles of Distributed Systems, pages 83–97. Springer, 2013.

3. Petrović, D., Ropars, T., and Schiper, A.: On the performance of delegation over cache-coherent shared memory. In Proceedings of the 2015 International Conference on Distributed Computing and Networking, page 17. ACM, 2015.

4. Fatourou, P. and Kallimanis, N. D.: A highly-efficient wait-free universal construction. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pages 325–334. ACM, 2011.

5. Hendler, D., Incze, I., Shavit, N., and Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures, pages 355–364. ACM, 2010.

6. Oyama, Y., Taura, K., and Yonezawa, A.: Executing parallel programs with synchronization bottlenecks efficiently. In Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, volume 16. Citeseer, 1999.

7. Yew, P.-C., Tzeng, N.-F., et al.: Distributing hot-spot addressing in large-scale multiprocessors. IEEE Transactions on Computers, 100:388–395, 1987.

8. Shalev, O. and Shavit, N.: Predictive log-synchronization. In ACM SIGOPS Operating Systems Review, volume 40, pages 305–315. ACM, 2006.

9. David, T., Guerraoui, R., and Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 33–48. ACM, 2013.

10. Roghanchi, S., Eriksson, J., and Basu, N.: Ffwd: Delegation is (much) faster than you think. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 342–358, New York, NY, USA, 2017. ACM.

11. Petrini, F., , Hoisie, A., Coll, S., and Frachtenberg, E.: The quadrics network (qsnet): high-performance clustering technology. In HOT 9 Interconnects. Symposium on High Performance Interconnects, pages 125–130, Aug 2001.

12. Pfister, G. F.: An introduction to the infiniband architecture. High Performance Mass Storage and Parallel I/O, 42:617–632, 2001.

13. Subramoni, H., Lai, P., Luo, M., and Panda, D. K.: Rdma over ethernet — a preliminary study. In 2009 IEEE International Conference on Cluster Computing and Workshops, pages 1–9, Aug 2009.

14. Peterson, C., Sutton, J., and Wiley, P.: iwarp: a 100-mops, liw microprocessor for multicomputers. IEEE Micro, 11(3):26–29, June 1991.

15. Mitchell, C., Geng, Y., and Li, J.: Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), pages 103–114, San Jose, CA, 2013. USENIX.

16. Dragojević, A., Narayanan, D., Castro, M., and Hodson, O.: Farm: Fast remote memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 401–414, Seattle, WA, 2014. USENIX Association.

17. Baenen, B. and Eriksson, J.: Trust¡t¿: A typesafe programming abstraction for delegation in rust. Technical report, University of Illinois at Chicago, Department of Computer Science, 2021.

VITA

NAME	Noaman Ahmad
EDUCATION	BS, Computer Science, Lahore University of Management Sciences, Lahore, Pakistan
TEACHING	Hands-on Rust: A Practical Introduction (CS194, Summer 2025)
PUBLICATIONS	