Ecole Polytechnique Fédérale de Lausanne

Data Center Systems Laboratory

R3P2: a Replicated Request-Response Pair Protocol

Projet de master, 2019, Section de Microtechnique

Author
Antoine Albertelli

Professor
Pr. Edouard Bugnion

Supervisor
Marios Kogias

Fall semester 2019

Acknowledgements

I would like to thank Marios Kogias for his help during all phases of this project. Thank you also to the whole DCSL/VLSC team for the interesting lunch time discussion and teachings.

Abstract

Consensus protocol have seen increased usage in recent years due to the industry shift to distributed computing. However, it has traditionally been implemented in the application layer. We propose to move the consensus protocol in the transport layer, to offer reliable ordered delivery of messages to applications.

Our implementation builds on R2P2, a novel RPC framework for datacenter applications. It can run both using Linux UDP sockets or kernel bypass using DPDK. We observe tail latency increase of only 10 µs and a throughput loss of only 13% compared to the unreplicated application, showing that easy-to-use consensus can also have high performance.

To Guido, Jocelyne & Ariane, for encouraging me to become who I am. As Ian M. Banks said, "My gratitude extends beyond the limits of my capacity to express it.".

Contents

1 Introduction **1**

2 Background **3**
2.1 Distributed Consensus 3
2.2 Transport protocols 6
2.3 Kernel Bypass 10

3 Design **12**
3.1 Choice of a consensus protocol 12
3.2 Modifications to R2P2 13
3.3 Request lifecycle 14

4 Implementation **16**
4.1 Architecture 16
4.2 Implementation language 17
4.3 Message Routing 17
4.4 Log 17
4.5 Worker threads 18
4.6 DPDK backend 18
4.7 Userland backend 19
4.8 Timers 19
4.9 Modification to client libraries 19
4.10 Implementation complexity 20

5 Evaluation **21**
5.1 Latency - Throughput 21
5.2 Impact of cluster size 22
5.3 Comparison with Kernel Paxos 22

6 Future work **26**
6.1 Disk backed persistency 26

6.2 Max-latency batching 26
6.3 Consistency verification of the protocol 27
6.4 Cluster membership change 27
6.5 Replicated request routing 27
6.6 Faster consensus protocol 28

7 Related Work **29**
7.1 Datacenter RPC 29
7.2 Kernel Bypass 30
7.3 Consensus in the network 31

8 Conclusion **32**

Chapter 1

Introduction

Since the introduction of computing systems, the amount of data to process have steadily increased. Historically, companies used larger and larger servers to cope with the increasing load. In recent years however, it became clear that switching to a large amount of unreliable, commodity servers was the most economical solution. Not only was the hardware for this approach cheaper, it also helped with availability. If a server component had to be replaced, it could simply be taken offline while the rest of the server handled requests.

Unfortunately, switching to this distributed model of computation also created new problems. The biggest of all was the one of data consistency: ensuring that node A and B had the same view of the world proved difficult. And when the network or machines in it are not reliable, it only becomes more difficult. This is called the *distributed consensus problem* and algorithms to solve it are part of almost any modern large scale system.

Traditionally, consensus algorithms have been part of the application layer and run on top of existing transport protocols. While this is easier to implement, it also means each distributed application must re-implement the consensus logic which is hard to get right. Imagine if each application had its own implementation of reliable, in-order transport?

From this observation, we decided to implement consensus at the transport layer. This means each application gets the possibility to become distributed “for free”. We implemented this as part of our Remote Procedure Call (RPC) oriented transport, which runs on top of UDP.

Our implementation is based on the Request Response Pair Protocol (R2P2), a novel transport layer developped at the Data Center Systems Laboratory (DCSL). This protocol was created specially with the goal of handling traffic inside data-centers. It embeds load balancing inside the protocol, providing different routing policies optimized for different load balancing methods. We added a new method called “replicated route”, in which all nodes are guaranteed to receive the same

message and in the same order (at the application level).

Our implementation can run either on top of UDP sockets or using custom userland networking on top of DPDK. While the former makes local development easier, the latter allows for high throughput and low latency by minimizing context switch overhead. We were able to get up to 275'000 messages per second on our test cluster of three machines, with 99th percentile latency as low as 30 µs (for an application processing time of 1 µs).

We also observed the effect of application workload on the performance of the consensus algorithm. In particular, we show that executing workloads in the networking thread on the replicas can lead to important losses of throughput. Therefore, our implementation executes the application logic in a separate thread from the networking and consensus code.

Chapter 2 provides some background about consensus algorithms and transports. Chapter 3 describes our protocol design, while Chapter 4 describes the reference implementation. Finally, Chapter 5 contains our evaluation of the system's performance under various loads. We also compare our consensus approach to other published ones in Chapter 7.

Chapter 2

Background

2.1 Distributed Consensus

When scaling out to large number of nodes, the likelihood of a node failing increases. Therefore, we would like the aggregate system to stay operational despite the failures of individual nodes. A very useful primitive to reach this property is called *distributed consensus*, which provides the cluster with a way to agree on a value despite individual failures.

This primitive can then be extended using asynchronous state machine replication. In this model, the values are instructions executed by a state machine running on each node. Since the initial state is specified, and all nodes in the cluster execute the same operations in the same order, we know that the state on every node after i instruction will be the same. Note that not all nodes might reach a given instruction at the same time. For example, a node separated from the peers by a network failure can not receive the latest update; it will catch up later, when connectivity is restored. However, once a value was chosen by the cluster, we know for sure that this value will eventually be accepted by all nodes. In other words, it is not possible for a node to change already-made decisions.

Typical replication protocols provide $2n + 1$ replication, meaning a cluster can stay alive as long as a majority of nodes is alive. For example, if there are 5 nodes in the cluster, up to two could go offline without impacting the availability or the consistency of the data. While it might seem like a good idea to make extremely large clusters, each additional node will increase the network load. A trade off between reliability and performance must, again, be made here.

An important restriction to make is the type of node failures that we are concerned with. We will be using the *non-byzantine* model of failures[1], in which the following properties hold. First, nodes may fail by stopping, and may restart (note that this requires persistent storage in case all nodes die). In particular, nodes

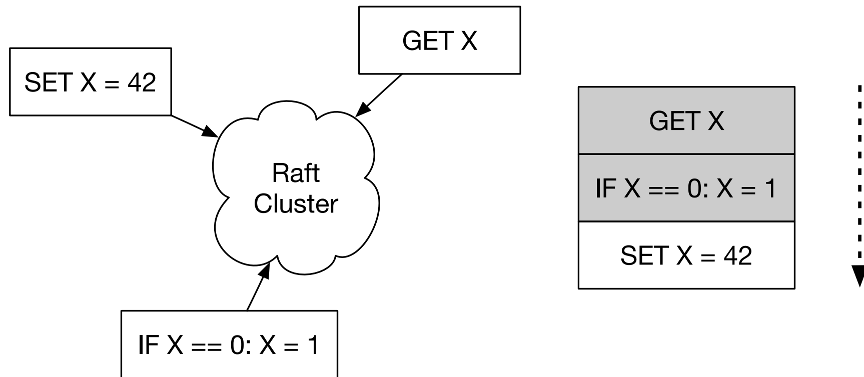


Figure 2.1: Example asynchronous operations for a simple key value store supporting atomic compare and swap operation. Also shown on the right is the replicated log. Entries in grey are committed, meaning they will never be overwritten, while the last one cannot be assumed to be persistent yet.

are assumed to be non-malicious and non-bogus. Messages can be re-ordered, can be duplicated, lost or delayed, but they cannot be corrupted. Dealing with those types of failure (especially malicious nodes) require a different set of algorithms, with significantly more complexity and less performance.

2.1.1 Raft Consensus Protocol

Raft[2] is a protocol that proposes a solution to the distributed consensus problem. Its main goal is to be easy to understand while still being efficient. It achieves its objective by cleanly separating different concerns of the protocol. It also provides a strong leader, *i.e.,* all entries flow from the leader to the followers, making it simpler to reason about. Raft provides a complete solution to build a replicated state machine; the original Paxos[3] only only provides agreement on a single value and requires an external leader election mechanism.

The approach chosen by Raft to model the consensus protocol is called a *distributed log*. Under this approach, the values replicated by Raft are put inside a queue called the log (Figure 2.1). Raft then replicates the log on all nodes, and guarantees that all the logs in the cluster will have the same operations in the same order. Raft also keeps track of which entries are replicated on every node. Once an entry has been replicated on a majority of machines, Raft guarantees it will never be removed from the log again. Such entries are called *committed*. Raft log entries can be used as instruction to implement a replicated state machine.

Raft can be split in three different parts: leader election, log entry replication and commit propagation. Each node can be in one of three states: *follower*, *candidate* or *leader*. During normal operation (when a leader emerged), a cluster

contains exactly one leader and zero candidates. To explain how the protocol works, we will start by examining the normal condition in which there is exactly one leader and it is operating properly. We will then explain what changes during leader election.

But first, we must introduce an important Raft concept called the *term*. One term is defined as a period of time in which there is at most one leader. This means that in order to change the leader, a new term must be changed. Terms are identified by their term number, which always increases. Note that not all terms have leader, for example if no leader could be elected, the term number is incremented before starting a new round of elections.

2.1.2 Log Replication

When the leader receives a request from a client, it appends it to its local log, tagging it with its index (a monotonic entry counter) and term number. It will then periodically send *AppendEntries* requests to the followers. Each of those contains all the entries that were not yet acknowledged by the destination follower. It also contains the term and index of the entries immediately before the first one in the request. This allows a follower to detect any gap between its local log and the incoming entries.

The destination follower will then append the new entries contained in the *AppendEntries* request to its own log. During this step, the entries' terms and indices are used to detect inconsistencies and duplicated entries. It then sends a reply to the leader to acknowledge that the incoming entries were correctly replicated. It can also notify the leader of a failure to replicate, for example due to a gap in the log.

Note that the checks performed when processing *AppendEntries* requests guarantee that, if two log entries on two different machines have the same index and term, then two properties must hold[2]. First, the two entries must store the same content. Then, all previous log entries are also matching between the two logs. Those consistency properties allow Raft to have a simpler entry committing mechanism.

2.1.3 Log Entry commit

We say that an entry is *committed* when we know for sure that this entry will never be lost by the cluster. We also know that a committed entry will eventually be replicated to the whole cluster. This means that to be committed, an entry must be replicated on a majority of servers. When an entry is marked as committed, its content can be consumed by the application.

Raft does not keep track of the commit status for each individual entry. Instead, it tracks the last committed entry; all entries before are considered committed too.

The leader keeps track of the latest acknowledged entry for each follower. Once an entry has been replicated on a majority of machines, the leader moves its commit index to it. The followers are then told to update their local commit status to the new index.

2.1.4 Leader Election

In Raft, leader election is based on timers. First, a periodic timer is used by the leader to send heartbeats to followers. Those heartbeats are *AppendEntries* requests, which can be empty if there is no new request to replicate. Every time one of those heartbeats is received, the follower resets the second timer, called the *election timeout* timer.

If no messages is received from the leader, then the election timer will fire. The follower can then then start a new leader election. It will first increment the term, as there can be only one leader per term. It will then transition to the candidate state and send *Vote* requests to other cluster participants.

Once a node receives a *Vote* request, it decides wether or not to grant its vote to the requesting candidate. To do so, it will first check that the candidate's term is more recent than its own. It will also check that the candidate's log is at least as complete as its own. This ensures that the leader's log contains all committed entries. If this was not the case, then the leader would start replacing committed entries, leading to loss of consistency. Finally, the node sends a reply to the candidate containing the status of its vote.

If a candidate reaches majority, then it transitions to the leader role and starts sending heartbeats. Otherwise, the election timeout will fire again, restarting the process at a new term. To avoid conflicting elections where no majority can occur, the timeout duration is randomized, so that election will eventually succeed. This procedure is summarized in Figure 2.2.

2.2 Transport protocols

Transport protocols build on the the network (IP) layer and provide services to the applications. The most important service is multiplexing; several application can use the network, and the transport routes information to them. This is done using port numbers. Historically, IP was used with two transport protocols: UDP and TCP.

While UDP is very simple and only distributes incoming packets to each application (via port numbers), TCP is much more complicated. It provides a

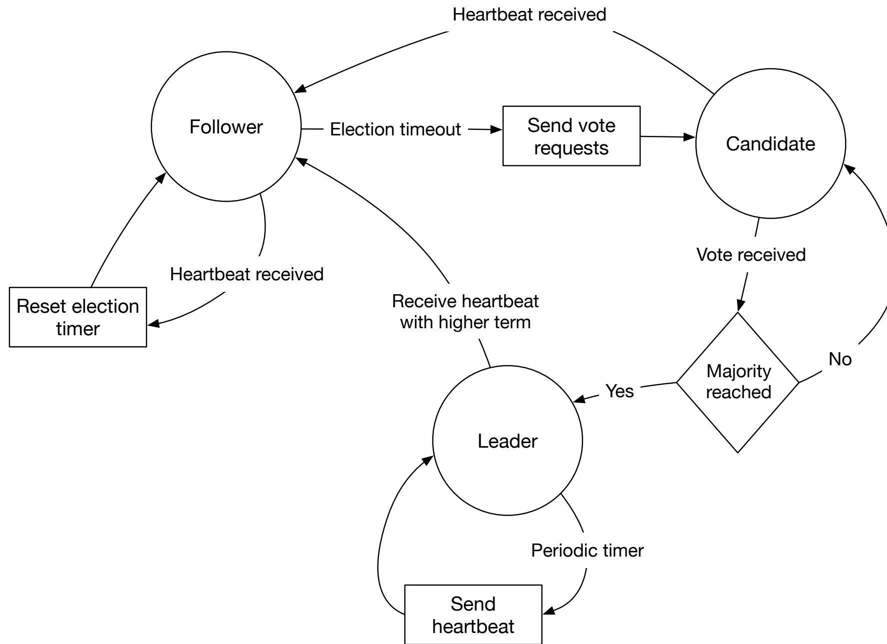


Figure 2.2: Raft node state machines showing leader election logic.

stream interface, which guarantees that bytes that enter the connection will arrive in the same order at the other end (reliable delivery). In order to do so, TCP has to establish a connection and then acknowledge packets. It also provides congestion control functionality to avoid link saturation.

All those features made TCP the logical choice when sending data on the Internet, which is pretty unreliable. However, when running inside a datacenter, packet loss or re-ordering is not as likely. In addition to this, TCP requires multiple round trips to track connection states before being able to send application data, which increases latency (Figure 2.3).

2.2.1 Request Response Pair Protocol (R2P2)

Most current RPC systems, such as Google's gRPC[4] or Facebook's Thrift[5] typically use TCP as their transport layer. In order to address TCP's shortcomings, the DCSL developped a new transport protocol specially designed for RPCs. Unlike TCP, this new protocol is connectionless; each communication is made of a single request followed by a single response, hence the name of Request Response Pair Protocol (R2P2). This reduces latency by removing the handshake RTT of TCP.

R2P2 has been succesfully used in the past to implement new load balancing techniques[6]. Since it was designed with load balancing, each R2P2 request includes includes a field to specify how it should be routed. For example, it can be marked as

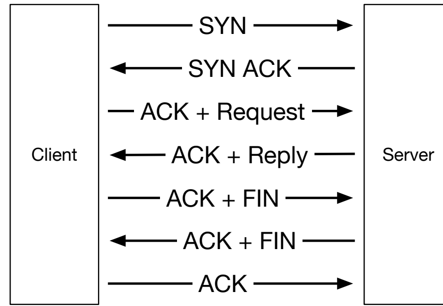
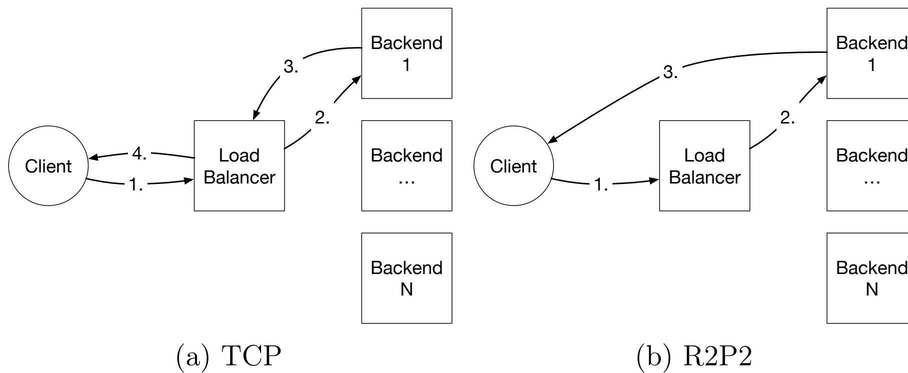


Figure 2.3: Lifecycle of a typical RPC over TCP (such as HTTP) interaction. We see that the latency before the reply is available takes at least two Round Trip Time (RTT).



(a) TCP (b) R2P2

Figure 2.4: Load balancing data flow using both TCP and R2P2. We can see that the connection-less nature of R2P2 allows data to be sent directly back to the client.

“Fixed”, meaning that the request must be served by the receiver, or “load-balanced”, in which case the router is free to redirect it to another server. R2P2 has no notion of connections, allowing the reply to be sent directly to the client instead of having an additional trip through the load balancer (Figure 2.4).

We realized that it could be interesting to embed the replication mechanism in the transport layer. This would greatly simplify the life of the application developer. Any networked application can be turned into a replicated version of it simply by changing a flag on the requests. Clients could even choose wether or not to ask for replication based on application-specific logic. For example, if stales read from a key-value store are acceptable, read request could be marked as “load-balanced”, while write requests would be marked as “replicated” to ensure consistency.

2.2.2 R2P2 API

The R2P2 library Application Programming Interface (API) looks nothing like a BSD socket API. Instead it is *event oriented*: the user of the library provides a set of callbacks that are called when a request is received (in the case of a server) or when a response arrives (in the case of a client). Those callbacks also receive a per-request argument which can be used to track state. The core of the R2P2 API can be seen in Listing 2.1 and 2.2.

Such event-oriented API maps well to the semantics of either high performance I/O syscalls (*e.g.,* Linux's `epoll`) or the one of kernel bypass frameworks such as DPDK.

```
/* Function types used for the various
 * callbacks */
typedef void (*success_cb_f)(
  long handle,
  void* arg,
  struct iovec* iov,
  int iovcnt);

/* Sending a request */
struct r2p2_ctx ctx = {0};
ctx.routing_policy = FIXED_ROUTE;
ctx.destination = /* snip */;

struct iovec local_iov;
local_iov.iov_base = msg;
local_iov.iov_len = strlen(msg) + 1;

ctx.success_cb = success_cb;
ctx.arg = &some_state;
r2p2_send_req(&local_iov, 1, &ctx);
```

Listing 2.1: R2P2 client API summary

With our proposal of bringing the consensus protocol to the transport layer, switching a normal application to a distributed, consistent one is simply a matter of changing the `routing_policy` field from `FIXED_ROUTE` to `REPLICATED_ROUTE`. We hope that this will create a reusable framework and that more developers will be able to write fault tolerant systems as a result.

```
void echo(long handle,
          struct iovec* iov,
          int iovcnt)
{
  struct iovec local_iov[1];
  memcpy(local_iov[0].iov_base, iov[0].iov_base);
  local_iov[0].iov_len = iov[0].iov_len;
  r2p2_send_response(handle, local_iov, 1);
}

/* ... */

r2p2_set_recv_cb(echo);
```

Listing 2.2: R2P2 server API

2.3 Kernel Bypass

Due to the way time sharing operating systems work, switching between userland and kernel code, or between two userland tasks is expensive: about 1 µs to 2 µs[7]). This is due to the need of saving the whole original execution context first, and then loading the new execution context. Since every system call switches to kernel code, high performance applications can be severly limited by the amount of syscall it is doing. For example, assume that a simple packet forwarding application running on Linux uses one syscall to read a packet, and another syscall to write it again. This means up to 4 µs of context switching per packet, restricting performance to about 250 kpacket/s.

Another important optimization used by high performance networking code is to avoid using interrupts. Interrupts are a way for a peripheral, such as a Network Interface Card (NIC), to notify the main processor, for example when an incoming packet is ready for processing. They allow the processor to avoid checking the presence of processing tasks needlessly, saving computational power and energy. However, they are also quite slow to be processed, increasing latency. In high packet rate applications, a processor can safely assume that a packet will always be available, or just check again if this is not the case. The loss in energy or computing power is quite low, and the gain in throughput and latency significant. By removing context switches and interrupts, Google observed a 3x gain in throughput in their load balancers[8].

To reduce the number of switches between kernel space and userland, we have two choices. The first would be to move more parts of the application in the kernel,

while the second moves more parts of the system to userland. The first option is the one that has been traditionnaly been used for networking; the TCP/IP stack or the filesystem are part of typical UNIX kernels. This technique is very efficient in terms of developer efforts; by using the kernel-provided facilities, engineers gain access to a high-quality implementation shared by many users. However, moving application-specific code in kernel space is not an easy task. This is due to the usual challenges of kernel development: very little memory protection, lack of debugging tools, possibility to crash the development machine if testing on it, *etc.* The lack of memory protection for kernel code also opens the door to security vulnerabilities, which can be pretty severe when processing network traffic. While modern operating systems offer security features to mitigate this risk[9], writing correct kernel code is still a difficult exercise. This is made even more difficult by the fact that Linux's internal API are considered unstable by developers. This means that applications may break as kernel gets updated, making maintenance more expensive than expected.

For all these reasons, modern high performance systems tend to move more, if not all, of the stack in userland, as part of the application. This technique is known as *kernel bypass*, because it bypasses the kernel to talk to the underlying hardware directly. In this approach, the application embeds everything it needs, from NIC drivers to TCP. While writing a NIC driver might seem like a waste of developer's time, one generally uses framework and libraries to implement this functionality. The main downside of kernel bypass is that it prevents sharing of ressources between applications running on the server. For example, each networked application would need its own NIC, which can be an issue in a commercial cloud environment. While this can be mitigated using some virtualisation techniques like SR-IOV, kernel bypass software is also more complicated to develop, and therefore reserved for performance sensitive applications.

A key contribution of our work is the use of kernel bypass techniques to reduce latency. In particular, we are opposing our design to Kernel Paxos[10], which moved the Paxos consensus protocol in the Linux kernel with great results. We believe that using kernel bypass techniques can lead to similar, if not better performance without compromising on the security benefit of process separation.

Chapter 3

Design

3.1 Choice of a consensus protocol

One of the earliest solution to the consensus problem is Lamport's Paxos algorithm[3]. It is currently widely used, for example Google relies on it to ensure correctness of its distributed systems[11, 12]. However, Paxos has two issues: first, it only solves part of the problem; some other "bricks" must be added to it to form a complete system. In addition to this, Paxos is known to be hard to understand, and even harder to implement correctly. Most of its users rely on pre-existing implementations such as libpaxos[1]. This means modifying the consensus protocol for our application and chosen transport could be hard to do

Fortunately for us, a simpler alternative to Paxos called Raft was recently introduced[2]. Raft was designed from the ground up to be simpler by cleanly separating the different parts of the problem. It also provides a complete solution, unlike Paxos which often requires additional, unproven extensions[12].

Another possible choice would have been ZooKeeper's ZAB[13], but it was not as well documented as Raft. In addition, it appeared to provide primitives that were less general purposes than a replicated log.

Despite the availability of excellent production grade Raft implementations, we decided to implement our own. Most existing Raft implementations are using high level languages such as Go, which are not suited for low latency programming. Other implementations in C or C++ make a lot of assumptions on the underlying platform. For all those reasons, it was deemed easier to do it ourself, using experience gathered by doing a prototype in Python. The resulting implementation is pretty small, at about 1000 lines of C++ code.

[1] `https://bitbucket.org/sciascid/libpaxos/`

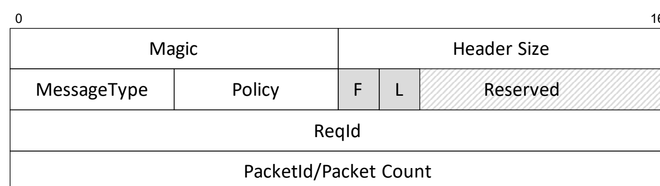


Figure 3.1: R2P2 header format[6]

3.2 Modifications to R2P2

R2P2 request headers have a *Policy* field to indicate to load balancers how this request should handled (Figure 3.1). The pre-existing policies were `LB_ROUTE` (can be redirected to any backend) or `FIXED_ROUTE` (the request should really be handled by the destination backend). We added the `REPLICATED_ROUTE` policy: all the backends will eventually receive this request. In addition, all backends are guaranteed to receive the requests using this policy in the same order.

Since this policy field is per request, it means clients can selectively enable request replication. For example, a user might specify that no write should be lost but that stale reads are acceptable (to lower latency). Then write requests would be sent with the `REPLICATED_ROUTE` policy, while reads are sent with `LB_ROUTE`.

Another addition to the R2P2 protocol was to add a new *MessageType* for Raft-related messages. The different types of Raft messages are differentiated in the R2P2 payload using a type field. The list of messages and their description can be found below. In addition, each message contains the term number of the sender, as it is required for all Raft operations.

VoteRequest Fields: `last_log_index`, `last_log_term`. Sent from one candidate to a follower to request a vote for itself. The candidate includes the last log entry's index and term so that followers can check if the candidate's log is up to date.

VoteReply Fields: `vote_reply`. Reply to a vote request, sent from a follower to a candidate. Contains a boolean indicating wether or not the follower granted its vote.

AppendEntriesRequest Fields: `count`, `leader_commit`, `previous_entry_term`, `previous_entry_index`, `entries`. This message is sent from the leader to the followers when there is new entries to add to the log, or periodically as a heartbeat if there is no activity. It contains data about the entry immediately before the provided one so that followers can check for gap in their log. It also serves to forward the commit index information to followers.

AppendEntryReply Fields: `success`, `last_index`. This message is sent from followers to leader and contains wether or not the new entries were successfully applied to the log, as well as the last index in the follower's log. This information can then be used by the leader to decide which entries to send next, as well as update the commit index.

3.3 Request lifecycle

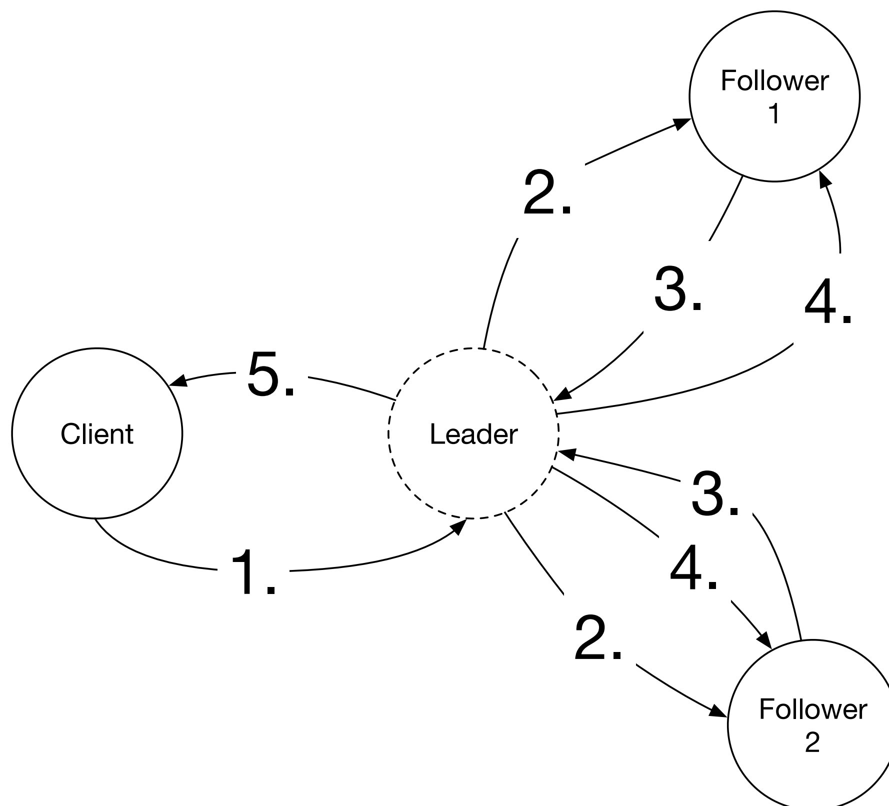


Figure 3.2: Typical client server interaction for a replicated service call. Request arrives to the leader (1). Request is then replicated to all followers (2). Followers acknowledge the replication (3). Leader marks the request committed (4). Commited requests are forwarded to the application and reply is sent to the client (5). Note that all arrows are complete R2P2 messages: if they span multiple UDP datagrams, we wait for the complete message to arrive before replicating it.

Figure 3.2 shows how a replicated request is handled. In this example, the first node is the leader, and therefore the request must be directed to it (1). The request is then packed in a log entry, and sent to the followers through Raft's

AppendEntryRequest message (2). Nodes then send back an *AppendEntryReply* message, acknowledging the message to the leader (3). Once a request has been acknowledged by a majority of the nodes, the leader marks it as committed. From this point on, the request will not be lost by the cluster[2]. Therefore, the request is forwarded to the application and the reply sent to the client (5). On next message, or on timeout, the new commit index will also be sent to followers (4). At this point they also forward it to their local copy of the application, which also sends back answers(5). The replies sent from non-leader are silently dropped, as they would arrive after the leader's reply.

[2]Provided that a majority of nodes stay healthy

Chapter 4

Implementation

4.1 Architecture

The global architecture of our system can be seen on Figure 4.1. At the top sits the application logic, which can be any general networked application: HTTP server, database, cache, *etc.* The implementation presented in this report can accomodate any of those.

The application layer uses services provided by the R2P2 library. The main role of this library is to assemble complete RPC messages from User Datagram Protocol (UDP) datagrams. It also handles the logic of replying to a given request, as well as some acknowledgements required for load balancing. This layer also provides networked I/O to the Raft implementation.

The R2P2 layer will in turn make calls into DPDK and Linux for various functionalities. We must emphasize that all data plane operations are done through DPDK, as calling into Linux is too slow for our purposes. Not shown here is also a small UDP/IP stack, with support for auxiliary features like the Address Resolution Protocol (ARP).

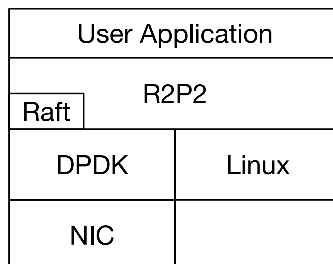


Figure 4.1: View of the different layers in the system architecture.

4.2 Implementation language

As the reference implementation of R2P2 was written in C, we wanted to use a language that could be easily integrated with the existing codebase. We decided to use C++ for the gain in expressiveness compared to pure C. In particular, templates allow code to be easily written in a generic fashion. Move semantics are also available in this language since C++11, and allow us to avoid a lot of packet copying while simplifying memory management. While not as useful as the one found in other languages, such as Rust, mostly due to the lack of safety guarantees, they still form a very useful tool to reduce packet copying while simplifying memory management.

4.3 Message Routing

In Raft, only the leader can process new client requests. This keeps the protocol simpler by only having data flowing from the leader to the followers. On the other hand, it means that we must also have a way to direct client queries to the leader.

In the original Raft paper[2], the followers will redirect clients to the leader if contacted directly. However, this is not optimal from a tail latency perspective: a client that picks a follower as a destination will see an additional RTT to its request.

Fortunately, R2P2 was implemented with load balancing semantics in mind. We can therefore extend the R2P2 load balancer to always propagate replicated requests to the current leader. While this haven't been done yet, it could be implementing by having the load balancer listening to Raft heartbeat messages (Figure 4.2).

For our benchmarking we manually pointed the benchmark client to the elected leader.

4.4 Log

At the end, Raft can only replicate a log, and ensure that all the nodes will eventually have all entries, in the same order. We must provide an adequate definition of the log entries for our needs. Since we are using Raft to replicate RPC requests, the log entries will contain a R2P2 request: A source address and port, a request ID, and the request data.

To keep things simple and low latency, our log implementation stores its data in RAM only. The requests are kept in a circular buffers, so that the oldest entry gets replaced by the newest one. The data is stored into the circular buffer without copying; when an entry is overwritten, its data is given back to the network stack.

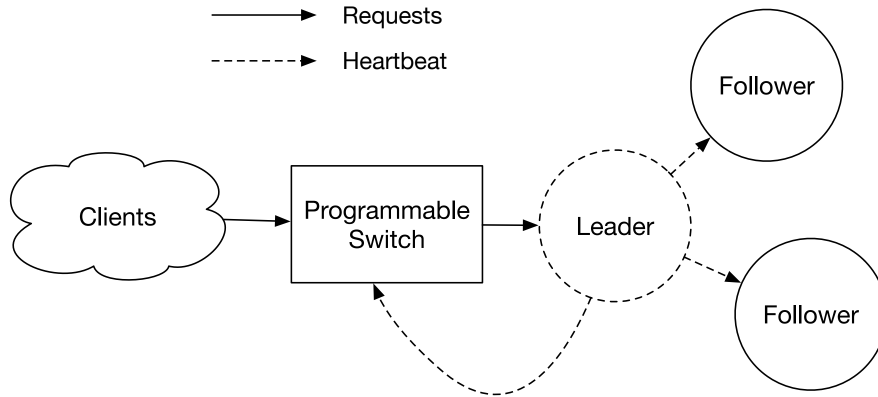


Figure 4.2: Message routing with R2P2 and Raft. The programmable switch listens to Raft leader heartbeats and forwards requests to the leader with the highest term.

4.5 Worker threads

During evaluation of the original design, we discovered that executing the application code in the main thread created a lot of queueing (see Section 5.1). Because of this, we decided to move the processing of the workloads off the main networking thread and into auxiliary workers. Data between the two is passed through queues.

However, simply using threadsafe queues would not have been enough. The overhead from mutexes can be quite important, reducing total throughput. The chosen approach is to use lockless queues, in which one reader thread can access the tail pointer, and one writer thread can access the head pointer. Since the pointers are not written by both threads and can only move in one direction (head cannot move back), it is safe to share the data structure without synchronization. We used an existing high performance implementation[14], as writing such data structures is error-prone and out of the scope of this work.

One downside of the current approach is that requests in the leader node are still executed by the main/network thread. This was done to avoid copying reponse data from one thread to another. However, it could cause some issues when a node transitions from follower to leader, if the application code is not thread safe.

4.6 DPDK backend

In order to provide the best performance, R2P2 implements kernel bypass. In this mode, the application talks directly to a dedicated NIC, embedding the driver in the library code.

For the implementation we opted to use Intel's DPDK framework. Several other kernel bypass framework exist such as netmap[15], or Ixy[16]. However they do not appear to have the same industrial traction as DPDK, neither are they cross platform; DPDK can run on Linux and FreeBSD on both Intel and ARM processors.

4.7 Userland backend

While kernel bypass offers the best performance, it is not always possible to use it (*i.e.,* when using a NIC is shared between applications). Our current implementation can therefore either be compiled to use DPDK (and a custom UDP/IP stack) or to use the normal kernel networking facilities.

To achieve high performance and stay close to DPDK's event based model, we use asynchronous I/O facilities. Our first implementation used Linux's `epoll` directly, making it non-portable. We rewrote it to use libuv[17] instead, which abstracts the different asynchronous I/O mechanisms on Linux, BSD and Windows.

This means that the current implementation can run as a normal networked component on all major platforms. This allows for a much easier development experience than running DPDK directly, and can even be used in production with good performance.

4.8 Timers

The Raft algorithm relies a lot on timers to detect leader absence and organize elections. In order to implement this, we simply use the functions provided by DPDK and libuv. The backend (DPDK or Linux) must only periodically call a function (`raft_tick`). This call is then used to process timeouts in various parts of the Raft implementation.

4.9 Modification to client libraries

On the client side, the code requires very little modifications to work with replicated request. One should just add the new routing policy definition. Nothing else is changed, and the client will still send and receive only one request response pair.

4.10 Implementation complexity

Our code can be divided in two big parts. The first one is the Raft logic implementation, which does not do any I/O, timer management, *etc.* It is about a thousand line of C++ code, which uses a lot templates and the standard library collections. In addition comes about 1500 lines of unit tests, very useful to check the soundness of our Raft code.

The second part is the part specific to R2P2, which implements custom log entries for R2P2 requests. This amounts to 250 lines of C++.

Finally we have a few changes to the core code in C, but they stay quite small. It mostly consists of routing requests tagged with `REPLICATED_ROUTE` to the Raft code.

Chapter 5

Evaluation

In this chapter, we measure the performance of our implementation. In particular, we will focus on two key characteristics: latency and throughput. Throughput is defined as the number of requests per second processed by the application. Latency is the time between the sending of a request and the reception of the reply (measured at the client). To generate the load and measure the latency, the Lancet distributed load generator[18] was used.

We designed two different experiments to explore how our system reacts to different conditions. In the first experiment, we measure the latency as a function of the achieved throughput. This will allow us to measure the peak throughput of the system, defined as the point at which the latency increases infinitely because requests are queued. This experiment was carried using a cluster of three nodes.

In the second experiment, we measured the impact of cluster size on latency. Since the leader must wait for a bigger quorum of machines to acknowledge the request, we expect the latency to go up with the number of machines. We also include the case where there is only one machine in the group, *i.e.,* when non replicated requests are used.

All experiments were done on machines equipped with two Intel Xeon E5-2650 CPUs running at 2.6 GHz. Each machine had 64 GB of RAM, out of which 2 GB were allocated to DPDK. The machines were equipped with Intel 82599ES 10 Gbit/s NICs connected through a switch. All the server machines were running the kernel bypass implementation of the stack, while the client was running the userland one.

5.1 Latency - Throughput

In this experiment, we varied the load on a three machine cluster cluster and measured the 99th percentile latency of the requests. The idea here is that the latency graph should exhibit a large vertical asymptote as we reach the maximum

capacity of the system. To have a baseline performance, we also ran the same load test on a non-replicated cluster.

Figure 5.1 shows that the latency of replica acknowledgment matters a lot for this type of system. Running the synthetic workload in the network thread (in a symmetric design) has very bad performance. For example, at a 200 µs latency Service Level Objective (SLO), running the workload in a separate thread has twice the throughput of the same-thread implementation. This was not what we expected, and had to be retrofitted in the implementation. This discovery should definitely be taken in account when designing future consensus systems.

5.2 Impact of cluster size

In this experiment, the goal was to see how the number of replicas impacted system performance. Adding new machines to the system increases its reliablity by increasing the number of failures required to have a loss of consistency. However, we can expect a loss of system performance when gaining in reliability. Throughput will go down, as adding replicas will increase the number of messages per request. It should also increase tail latency, but not by much, as we must wait on a bigger quorum of machines. The results can be see on Figure 5.2.

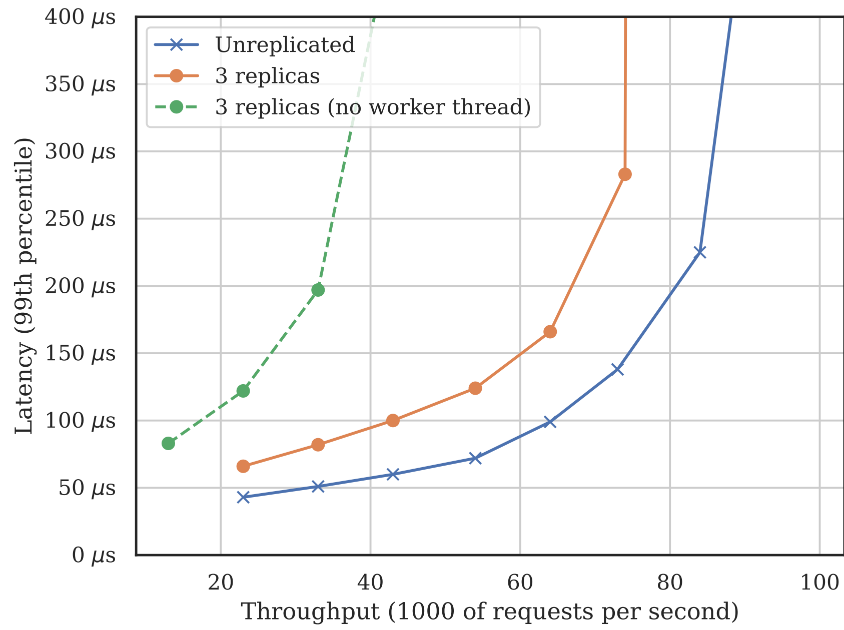
5.3 Comparison with Kernel Paxos

Kernel Paxos[10] is an earlier attempts to reduce consensus protocol latency by removing the cost of context switching. In order to do so, they implemented the Paxos protocol as a set of Linux kernel modules. They had to port the libpaxos implementation, which was intended to run in userland, to things like kernel memory allocation. Here are the main differences between their approach and ours:

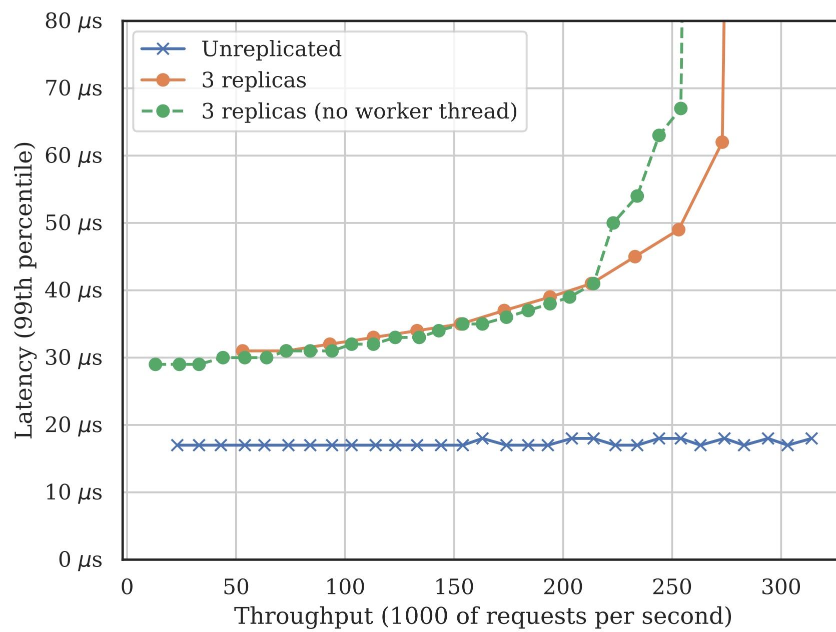
- Kernel Paxos is implemented as a set of Linux kernel modules. Our implementation uses Intel's DPDK[19] to access the NIC from userland.
- Kernel Paxos uses the Paxos algorithm. Our solution uses Raft.
- Kernel Paxos uses raw Ethernet frames to send its messages. This requires all machines to be in the same broadcast domain. Our implementation runs on top of UDP/IP and therefore can be routed.

The Kernel Paxos source code is freely available on the internet[1]. Unfortunately we were not able to run it on our experimental setup. Therefore, direct performance

[1] `https://github.com/esposem/Kernel_Paxos`

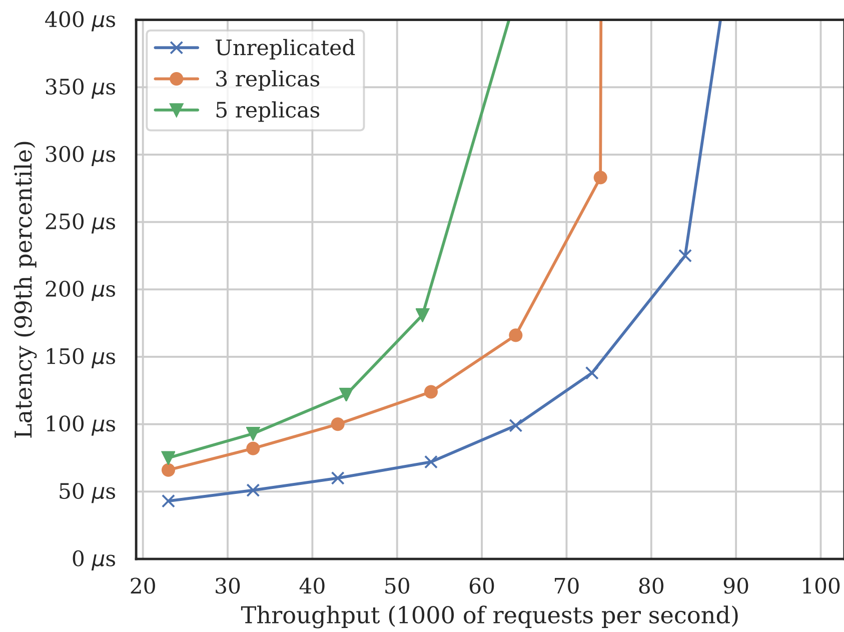


(a) 10 µs synthetic workload

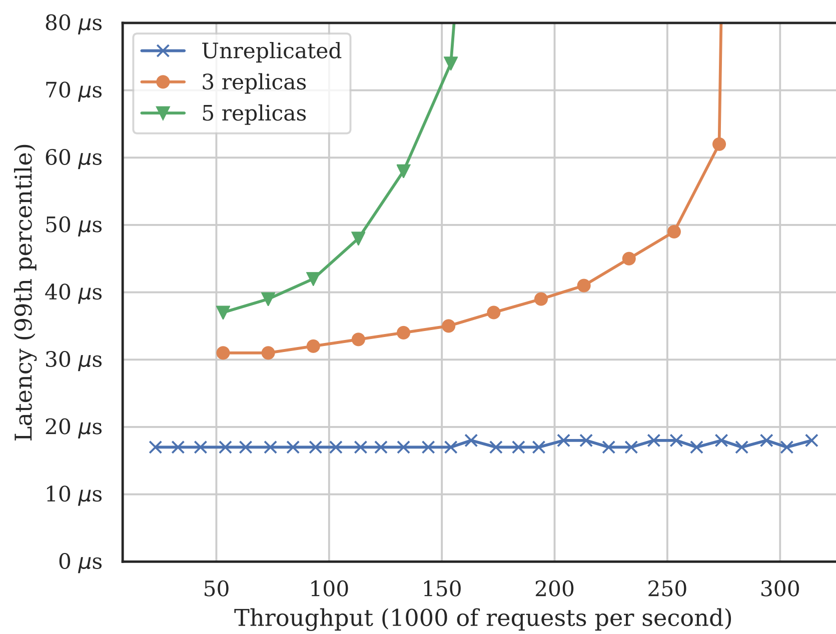


(b) 1 µs synthetic workload

Figure 5.1: Throughput of the system both with followers executing application code both in the network thread and in a separate thread.



(a) 10 µs synthetic workload



(b) 1 µs synthetic workload

Figure 5.2: Impact of quorum size on system performance. This experiment was performed using the worker thread implementation.

comparison might not be very accurate. In particular, their node's CPU is much older than ours (2008 vs 2012).

The most interesting result is latency: Kernel Paxos claims to have a median latency of 52 µs. We compare this with our 1 µs synthetic workload, which is closer to the Kernel Paxos case (which does not do any computation on the chosen values). We achieve a tail latency of 31 µs, despite a Paxos optimization called Fast Paxos[20]. In Fast Paxos, clients can bypass the leader, saving one network hop.

While Fast Paxos could be theoretically applied to our system, we decided not to for several reasons. First, we already had a working Raft implementation at this point in our work, and Fast Paxos is significantly harder to implement than Paxos. Then, Fast Paxos requires more machines to reach consensus: 2/3 of the nodes must be healthy for a value to be accepted. This means that to handle n failures, Fast Paxos requires $3n + 1$ replicas, where classic Paxos and Raft only require $2n + 1$. Note that if the $3n + 1$ replica conditions is not met, Fast Paxos will fall back to classic Paxos. Finally, under high loads, the probability of message collision is quite high, and in case of collisions the number of network hops in Fast Paxos is the same as with Classic Paxos or Raft.

On the throughput side, Kernel Paxos claims 175 kmsg/s, whereas our implementation reached 275 kmsg/s in a three machines cluster, a $1.6x$ increase. According to Kernel Paxos authors, their throughput is bound by CPU performance on the proposers. We were not able to verify this claim, as we were not able to run their implementation in our cluster. As they did not publish the model of their CPU for their 10 Gbit/s experiments, a direct comparison is hard to do.

In terms of implementation complexity, Kernel Paxos is about 8800 lines of code. Our implementation is about half this size at 4300 lines. In order to keep the comparison meaningful, the userland UDP/IP stack is not accounted in those counts.

Chapter 6

Future work

6.1 Disk backed persistency

The current implementation only stores requests in main memory. This means that requests are not truly persistent: if all nodes were to come offline, committed data would be lost. To avoid this, the replicated log would need to be persisted to non volatile storage.

However our application cannot use the standard Linux filesystem API due its latency cost. Recently new technologies have emerged to use persistent storage (particularly Solid State Disks (SSDs)) without relying on the kernel. One of those approach is Intel's SPDK[21], which uses kernel bypass techniques and presents an API similar to DPDK. Another option would be to use the newly introduced Linux's `IOCTX_FLAG_SQTHREAD` flag for polled IO[22]. This works by submitting filesystem operations in a ring buffer, which will get collected and executed by the kernel later.

6.2 Max-latency batching

Currently, our Raft implementation immediately sends out *AppendEntries* messages as soon as an incoming request is received. While this optimal from a latency perspective, it creates a lot of overhead if most requests are small. The classic way to circumvent this problem is to batch requests before sending one *AppendEntries* message (*i.e.,* like the original Raft implementation[2]).

The downside of naive batching like this is that it increases latency. An alternative might be to choose the batching period according to some SLOs; if some requests are more latency-sensitive than others, the latency SLO could be included in the R2P2 header itself, allowing the server to take an informed batching decision. We could also implement adaptive batching, in which batching is only

applied during congestion periods, minimizing the impact on latency, as done in the IX dataplane operating system[23].

6.3 Consistency verification of the protocol

The current implementation of the protocol has been mostly tested using hand-written unit and integration tests. However, Raft is hard to implement correctly, despite being designed for ease of comprehension.

Jepsen[24] is an automated tool for checking of consistency properties in distributed systems. It works by randomly applying partitions, delays and packet reordering to distributed applications. So far it sucessfuly analyzed and found issues in major projects such as MongoDB, etcd, or Redis. We think we would gain a lot of confidence in the soundness of our implementation if a Jepsen testbench was available for it. It could be written using R2P2 as its only interface, so that the same test suite could be shared between different implementations.

6.4 Cluster membership change

An important limitation of the current implementation is that cluster size cannot be changed without restarting the application. This is an important restriction, as it prevents scaling the system or replacing failing nodes. The Raft paper[2] proposes a way to deal with the addition or removal of replicas.

The main challenge comes from not wanting to have two leaders at any given point in time. This could happen if the quorum size changes, and not all nodes receive this information. Then two differente leaders could lead to two different values being accepted, without a way to reconciliate those. In order to avoid this, a new mechanism is introduced called *join consensus*, where all majority agreements (entry commit and leader election) must be reached both in the old and new quorums. A catch-up phase is also needed, in which new cluster members are downloading entries from the leader and do not take part in the vote process.

This was kept out of this work because it was not needed to prove that moving replication in the transport worked. However, it would be a very useful extension for real-life deployements. All the required details can be found in Section 6 of the Raft paper[2, p. 10].

6.5 Replicated request routing

As mentionned in Section 4.3, messages under Raft must go through the leader, but clients do not know who the leader is. A solution would be to have a smart switch

listen to Raft leader heartbeats to decide who should receive messages tagged with `REPLICATED_ROUTE`. Since a R2P2 load balancer is already implemented in the P4 programming language[6], the Raft-related functionality could be added to it with excellent performance.

6.6 Faster consensus protocol

Our current consensus implementation is very generic: it makes no particular assumptions about the properties of the underlying network. This is good for Internet routing or public clouds, where we control very little about the network fabric. It also means we get consensus on 2 RTTs.

However, in typical datacenter deployements, we can modify the network to support the applications. Things like Quality of Service (QoS), VLANs and multicast group offer new functionalities that the applications can rely on. Using this mechanism, Ports *et al.* used a single network switch as a serialization point for all multicast traffic. They were able to design a consensus protocol that reaches consensus in one RTT in 99.9% of requests called *Speculative Paxos*[25].

One downside of this approach is that some transactions sometimes need to be rolled back. This would require exposing a more complicated API to the application layer. However, this would be doable to reach ultra low latency consensus.

Chapter 7

Related Work

7.1 Datacenter RPC

R2P2 was not the first alternative transport for datacenter RPC. Over recent years, several alternatives have been suggested, based on the increased scaling difficulty of existing protocols.

eRPC[26] is a novel RPC protocol for datacenter applications. Like R2P2, it can be deployed on conventional Ethernet/IP fabrics for existing network compatibility. However, it can also be deployed on lossless fabrics like Infiniband, which might become more relevant in future datacenter designs. Their design provides end to end flow control (like TCP) by using credits on the sender side: each sender spends one credit when sending a packet, and gets one back when receiving one packet. eRPC also provide congestion control, but optimizes for the common case of an uncongested network. On a 100 Gbit/s network, eRPC can get median latencies as low as 2 µs, close to an harware-assisted Remote Direct Memory Access (RDMA) read (2.3 µs).

Raft was implemented as an application running on top of eRPC. Their experiment shown a replication tail latency of 6.3 µs on 100 Gbit/s hardware. We expect our implementation to have similar results when running on similar hardware, but we did not test this.

Another transport protocol design is Homa[27], which only targets conventional Ethernet deployements, like R2P2. The key observation driving Homa's design was that congestion control is very bad for latency of short messages, increasing the packet latency by a factor 3 (0.5 RTT to 1.5 RTT) to contact a coordinator or the receiver. However, congestion in the network buffers will also negatively impact tail latency. To solve this problem, Homa uses hardware priority queues to prioritize packets with the shortest remaining data transmission. The priorities are allocated by the receiver based on the total message size (contained in the header) and sent

back to the senders. Homa also reduces incast by keeping track of the number of in-flight RPCs, which reduces network congestion further.

7.2 Kernel Bypass

IX[23] and Arrakis[28] are both recent specialized kernel bypass frameworks. They both come from the observatoin that the overhead of having the Linux operating system in the data path causes too much latency due to complex software path and costly context switching between kernel and application. However, they also acknowledge that some of this latency is due to security and isolation functionalities, which have historically been implemented as part of the kernel. In order to keep applications isolated, IX and Arrakis use Intel's virtualization extensions to isolate the different data planes from each other and from the control plane. While IX uses Linux as the virtualization supervisor and control plane, Arrakis is based on a research operating system called Barrelfish. Both IX and Arrakis implemented custom system calls after observing that POSIX semantics were a poor fit to NICs hardware queues. Arrakis also tackles the storage space, providing kernel bypass for Redundant Array of Independent Disks (RAID) controllers.

Arrakis and IX implement a run-to-completion model, in which all the processing for a packet (or batch of packet) is done before moving to the next one. This increases performance by having better cache locality and avoiding copies. However it can also lead to unbalanced situations where some CPU cores are idle, while others still have incoming packets enqueued, especially if task duration vary a lot. ZygOS[29] addresses this issue by having a queue between the network layer and the application layer. A scheduler then implements task stealing, allowing idle CPU cores to take tasks from other cores. This breaks from the run-to-completion model by adding a queue in the middle. Authors note a 1.26x speedup over IX and 1.63x over Linux.

NetBricks[30] takes a different approach for data plane isolation. Where Linux enforces this through the kernel and IX through hardware extensions, NetBricks proposes the use of compile-time checks for safety. Their implementation uses Rust, a modern system language. Rust's memory ownership semantics, enforced at compile time, allow them to implement most operation in a safe and zero-copy way. They observe impressive speedups, especially when chaining dataplanes: up to 7x throughput gains compared to container isolation.

7.3 Consensus in the network

As the industry moves to Software Defined Networking (SDN) inside datacenter, programmable switches become more available and targetting them for application logic becomes reasonable. Several projects tried to implement Paxos rounds as part of the network to achieve better performance than userland implementations. Dang *et al.* wrote a P4 implementation of the Paxos leader and acceptor roles[31], while the clients (proposers and learners) are still implemented on commodity servers. They do not provide a benchmark of their work, which is centered on implementation strategies. In particular, they implemented standard Paxos, but note that Fast Paxos or Speculative Paxos could be applicable here.

Jialin *et al.* propose a new networking primitive, Ordered Unreliable Multicast (OUM), which guarantees that all nodes in the multicast group who receive the packets will do in the same order[32]. However, OUM does not guarantee packet delivery; some nodes might lose packets, which will create a gap in their log. OUM is implemented using SDN rules to route all traffic to the multicast group through a single programmable switch, which acts like an ordering point and writes a sequence number into packets. NOPaxos uses the ordering provided by OUM when there is no packet drop and fall back to classic Paxos in case of packet loss. The author observe performance very close to an unreplicated system, but note that implementing OUM in a programmable switch rather than a software middlebox could close the gap even further.

Another challenge where replication could be useful is in storage. Recently, Storage Class Memories (SCM) became available, offering ultra low latencies and high throughput. However, they come with a unique set of challenges, in particular they are prone to storage cell wear, making them less reliable than conventional storage.

To fix this reliability issue, one option would be to use a replicated set of memories. As the problem of writing and reading to memory atomically is less general than a replicated log, a simpler, more efficient algorithm called ABD[33] can be used. A demo was implement using Linux on the client side and FPGAs on the server side[34]. Their implementation was able to read a replicated cache line in about 18 µs vs about 3 µs for one stored in local RAM.

Chapter 8

Conclusion

We presented an extension to the R2P2 transport protocol that offers reliable, in-order, delivery of messages to application. Our implementation uses well-known techniques like kernel bypass to offer both low tail latencies and high throughput. We validated our design by running it against a series of microbenchmarks simulating different application workloads. We observe an additional tail latency of only 13 µs compared to the unreplicated case. On the throughput side, we only lose 14% at a given SLO. This shows that our original idea, moving consensus to the transport layer, can be both a general and high-performance tool for the distributed application developer.

One key finding from this work is that latency of follower nodes matter a lot. This means that application code should run on a separate thread on followers, to ensure the network code is not waiting on slow application requests. We observed a throughput gain of up to 2x when implementing this separation of threads.

We believe that moving the consensus to the transport layer is the way to go for easy development of distributed applications. However, before this solution can be used, there are some challenging issues to solve. In particular, log compaction and persistent storage must be implemented to ensure true reliable consensus. This could be a good extension to this project for a future master thesis.

Abbreviations

API Application Programming Interface. 9–11, 26, 28

ARP Address Resolution Protocol. 16

DCSL Data Center Systems Laboratory. 1, 7

NIC Network Interface Card. 10, 11, 18, 19, 21, 22, 30

OUM Ordered Unreliable Multicast. 31

QoS Quality of Service. 28

R2P2 Request Response Pair Protocol. 1, 7–10, 13, 14, 16–18, 20, 26–29, 32

RAID Redundant Array of Independent Disks. 30

RDMA Remote Direct Memory Access. 29

RPC Remote Procedure Call. 1, 7, 8, 16, 17, 29, 30

RTT Round Trip Time. 7, 8, 17, 28, 29

SCM Storage Class Memories. 31

SDN Software Defined Networking. 31

SLO Service Level Objective. 22, 26, 32

SSD Solid State Disk. 26

UDP User Datagram Protocol. 16

Bibliography

[1] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[2] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm.," in *USENIX Annual Technical Conference*, pp. 305–319, 2014.

[3] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer networks*, vol. 2, no. 2, pp. 95–114, 1978.

[4] "gRPC, A high performance RPC framework." `https://grpc.io/`, 2015.

[5] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," tech. rep., Facebook, 2007.

[6] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 863–880, USENIX Association, 2019.

[7] E. Bendersky, "Measuring context switching and memory overheads for linux threads." `https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/`, 2018.

[8] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer.," in *Proceedings of the 13th symposium on Networked systems design and implementation*, pp. 523–535, USENIX Association, 2016.

[9] The kernel development community, "Kernel self-protection." `https://www.kernel.org/doc/html/v4.18/security/self-protection.html`, 2018.

[10] E. G. Esposito, P. Coelho, and F. Pedone, "Kernel Paxos," in *Reliable Distributed Systems, 2018 IEEE International Symposium on*, IEEE, 2018.

[11] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335–350, USENIX Association, 2006.

[12] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pp. 398–407, ACM, 2007.

[13] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*, vol. 8, Boston, MA, USA, 2010.

[14] C. Desrochers, "A Fast Lock-Free Queue for C++." `http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++`, 2013.

[15] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 101–112, 2012.

[16] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle, "User space network drivers." `https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ixy_paper_draft2.pdf`, 2018. Draft.

[17] The libuv team, "libuv." http://libuv.org/, 2013.

[18] M. Kogias, S. Mallon, and E. Bugnion, "Lancet: A self-correcting latency measuring tool," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 881–896, USENIX Association, 2019.

[19] "DPDK, The Data Plane Development Kit." `https://www.dpdk.org/`.

[20] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[21] "SPDK, The Storage Performance Development Kit." `https://spdk.io/`.

[22] J. Axboe, "aio: enable polling for IOCTX_FLAG_SQTHREAD." `https://patchwork.kernel.org/patch/10740953/`, 2018.

[23] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI)*, USENIX, 2014.

[24] K. Kingsbury, "Jepsen: Distributed systems safety research." `https://jepsen.io/`, 2013.

[25] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing Distributed Systems Using Approximate Synchrony in Data Center Networks.," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 43–57, USENIX Association, 2015.

[26] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), USENIX Association, 2019.

[27] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, (New York, NY, USA), pp. 221–235, ACM, 2018.

[28] S. Peter, J. Li, I. Zhang, D. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, p. 11, 2016.

[29] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 325–341, ACM, 2017.

[30] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV.," in *Proceedings of the 12th symposium on Operating systems design and implementation*, pp. 203–216, 2016.

[31] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, 2016.

[32] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. Ports, "Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering," in *Proceedings of the 12th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 467–483, 2016.

[33] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 124–142, 1995.

[34] H. T. Dang, J. Hofmann, Y. Liu, M. Radi, D. Vucinic, R. Soulé, and F. Pedone, "Consensus for non-volatile main memory," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pp. 406–411, IEEE, 2018.