Inter-process Communication in Disaggregated Datacenters

by

Amanda Carbonari

BSc. Computer Science, Colorado State University, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2018

Abstract

Disaggregation is a promising new datacenter (DC) architecture which aims to mitigate mounting DC costs. Disaggregated datacenters (DDCS) disaggregate traditional server components into distinct resources. Disaggregation also poses an interesting paradigm shift. Namely, a DDC possesses traits akin to a distributed system, as resources no longer fate- share: a CPU can fail independently of another CPU. It is not unreasonable to assume that these disaggregated resources will still be presented to a user as a single machine. This requirement has implications for disaggregated system design. For example, what happens if a CPU fails during a remote cross-processor procedure call?

This is not a new question, as distributed systems, multi-processor systems, and high performance computing (HPC) systems, have grappled with this challenge. We look at how this challenge translates to a disaggregated context, in particular, focusing on the remote procedure call (RPC) abstraction. We design a disaggregated system, Bifröst, to ensure *exactly-once* semantics for procedure calls under failure scenarios and provide strict memory consistency. We analyze the overhead of Bifröst compared to an equivalent RPC implementation in Thrift. Although, we find that Bifröst has a higher overhead than Thrift, its results are still promising, showing that we can achieve greater functionality than Thrift with a slightly higher overhead.

Lay Summary

Datacenters are costly to operate due to cooling costs, machine upgrades, etc. Disaggregation, a trend of separating resources into individual entities, attempts to mitigate these costs. Once the resources are separated, the datacenter no longer provides the same single machine architecture users typically work with.

To provide this single machine abstraction, the disaggregated datacenter must provide some guarantees about the resources. In particular, what will happen if the compute resource of the machine fails? We focus on providing memory consistency and communication guarantees even under failure for disaggregated systems.

Preface

The work presented in this thesis was conducted by the author in collaboration with Fabian Ruffy under the supervision of Dr. Ivan Beschastnikh. None of the text of the dissertation is taken directly from previous published or collaborative articles.

The system design in Chapter 5, inter-process communication system in Chapter 6, and the experiments in Chapter 7 were primarily implemented and performed by me. Design choices for Chapter 4 and Chapter 5.3 were influenced by feedback and input from Ivan Beschastnikh and Fabian Ruffy. The implementation of the switch and distributed shared memory system in Chapter 6 was primarily done by Fabian Ruffy.

Table of Contents

List of Tables

List of Figures

Glossary

RPC remote procedure call

DC datacenter

DDC disaggregated datacenter

HPC high performance computing

TOR top of rack

IPC inter-process communication

PISA protocol independent switch architecture

DSM distributed shared memory

API application programming interface

OVS OpenV Switch

SDN software defined networking

RDMA remote memory access

RTT round trip time

ICMP Internet control message protocol

MSS maximum segment size

MTU maximum transmission unit

Acknowledgments

This research is supported by an NSERC discovery grant. I would like to thank my supervisor, Ivan Beschastnikh for shepherding me through my Masters and helping to grow and improve as a researcher. I'd also like to thank Fabian Ruffy for all the work he put into the project, as this thesis would not have been possible without his contributions. Finally, I'd like to thank my finacé, parents, and NSS lab-mates who supported me throughout my Masters.

Chapter 1

Introduction

Disaggregation is a rising trend that attempts to mitigate mounting datacenter operational costs [21]. Disaggregated datacenters (DDCS) separate the traditional resources of a server-centric architecture into individual components. A *blade* is a server which consists of one specific *resource* type (i.e., CPU, memory, SSD, etc.), each individual resource is connected over a commodity interconnect (e.g., Ethernet). This architecture provides many benefits to both users and operators, chief among them are modularity and density [16].

But, disaggregation also poses an interesting paradigm shift. Namely, a DDC possesses traits akin to a distributed system as resources no longer fate-share: a CPU can fail independently of another CPU. Yet, it is reasonable to assume that disaggregated resources will be compiled and presented to the user as a single machine to support legacy applications [10, 16].

Datacenters currently support a variety of applications and workloads. Ideally, these applications and workloads will continue to have the same behavior on DDCS. Yet, a legacy datacenter application, such as Hadoop, cannot reason about the memory of a node failing or one processor in a node failing. Therefore, a DDC must provide strict guarantees to legacy applications when components fail.

But, how can the underlying system abstract away CPU failure during a remote cross-processor procedure call? This particular question has been explored in other areas of research, such as high performance computing (HPC), multi-processor systems, and distributed systems.

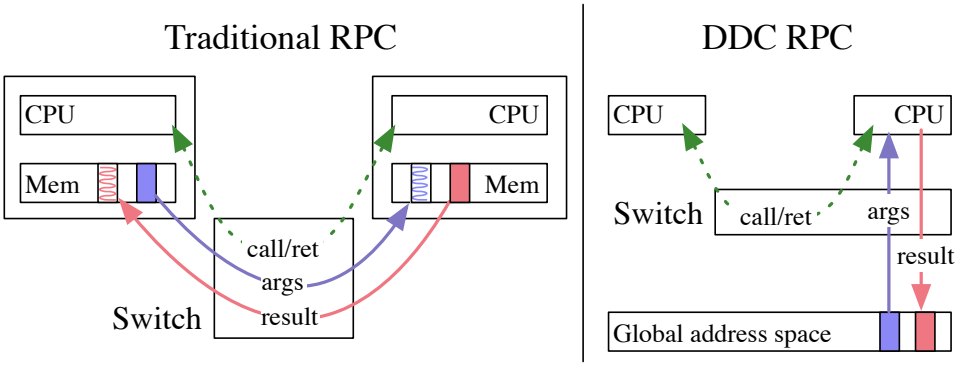

Figure 1.1: Pass by reference semantics for DDC RPC compared to traditional RPC.

In particular, remote procedure calls (RPCS) have been used in both distributed systems and multi-processor systems for inter-process communication (IPC). Unfortunately, current RPC implementations are limited due to failures and inability to use pass by reference arguments. There are multiple practical challenges in attempting to achieve local procedure call semantics (*exactly-once*) under failure conditions and reason about memory addresses on remote nodes [8].

We focus on the challenges of building reliable RPC mechanism for disaggregation. Disaggregation allows for optimizations previously unavailable in distributed systems, primarily, a global shared memory space. Since DDC resources are compiled to represent a single machine, each processor in a DDC "machine" has uniform access to a bank of global memory. We leverage this global address space to support pass by reference arguments (Figure 1.1).

Rack-scale DDCS also differ from distributed systems and multi-processor systems because most datacenter racks today incorporate a top of rack (TOR) switch. We find that the TOR switch presents a natural interposition point to observe cross-processor procedure calls and memory accesses. This allows for a control-plane program that monitors all procedure calls and pointer arguments. This program then interfaces with daemons running on the resources to coordinate and enforce fate-sharing and failure recovery.

We prototype our design, Bifröst, in an emulated disaggregated rack setting. We evaluate the overhead of our system relative to an equivalent Thrift RPC imple-

mentation [3].

In summary, our work makes the following two contributions.

- We define an IPC semantics for a disaggregated rack (Section 4).
- We present Bifröst, a system co-designed with the network, that ensures *exactly-once* semantics of procedure calls even under failure and enforces strict memory consistency (Section 5).

We describe our prototype of Bifröst in Section 6 and evaluate it, exploring the performance implications relative to current RPC implementations (Section 7). We conclude with the limitations of our work and future directions (Section 8).

Chapter 2

Motivation

Bifröst is motivated by the recent trend in datacenter architecture and programmable switches. We argue that, to reap the full benefits of disaggregation, we must take a holistic approach to designing systems on disaggregated racks. In particular, we must consider the role of the network as a critical piece in the design. In this work, we focus on the ability of the network to provide fault tolerance and fate-sharing enforcement at line rate, ensuring exactly-once semantics for procedure calls, as well as enforce strict memory consistency semantics.

2.1 Failure rates in large-scale systems

As systems begin to scale beyond traditional servers into large rack-scale machines, we must consider the implications of failures. HPC has been studying failure rates in supercomputers for over a decade. They have found that failures not only occur at a rate which requires mitigation, but also at a high enough rate to have detrimental effects on performance [15, 36, 38, 40, 43]. Although disaggregation is not at the same scale as supercomputers, it is progressing in that direction. Disaggregation also has the same requirement as supercomputers: a conglomeration of resources must be presented as a single entity to the programmer. Therefore, failures must not only be addressed in disaggregation, but they must also be mitigated at the system level due to performance implications.

2.2 Strawman argument

A tempting strawman argument is to enforce traditional fate-sharing semantics. Essentially making DDCS fail like traditional servers. This will lead to inefficient use of resources under failure [10]. Disaggregation allows for new fate-sharing models and possibly new fault- tolerance techniques, these must be discovered and implemented on a per application basis. In our particular scope, we look beyond fate-sharing between caller and callees, although our system still provides it. We design a way to recover from caller and callee failure instead of failing functional processes.

2.3 Existing architectures

Disaggregation presents a different context than many of the existing similar architectures that handle inter-process communication failures (distributed systems, multi-processor systems, and HPC).

Disaggregation differs from distributed systems because distributed systems maintain a "node" view, where a collection of resources (CPU, memory, storage) is lost when a "node" fails [20]. Disaggregation must contend with individual resources failing [10]. Distributed systems also deal with more points of failure than a disaggregated rack. For example, distributed systems must handle network failures such as partitioning and packet loss [20]. Due to these differences, distributed system solutions do not take advantage of the disaggregated resources and make trade-offs to cover failures not present in disaggregation.

Multi-processor systems differ because they were built on a smaller scale with static configurations [7, 11]. Disaggregation must contend with failures at the rack-scale, but can also replace components with free resources in the rack. Multi-processor systems were also built using intra-connects between resources, not commodity, packet-switched networks. Therefore, disaggregation is a less constrained environment than multi-processor systems, and must expand up on their solutions to match the elasticity and scale of the environment.

Although HPC systems match if not exceed disaggregation in scale, they rely heavily on specialized hardware [14]. It is reasonable to assume that disaggregated racks will still be built from commodity hardware to mitigate costs. Therefore

they cannot rely on specialized hardware for solutions. HPC systems often require specific programming paradigms from their users. This is not possible in a DDC because datacenters today serve a variety of workloads, from hosting web servers to large-scale distributed data processing. Programmers cannot be constrained in the workloads they can run in the datacenter.

Each of these areas differ from disaggregation in a variety of ways, but we can learn from these solutions and port them to the disaggregated context. We attempt to do so while focusing on IPC.

2.4 Programmable switches for resource management

The TOR switch provides a natural interposition layer for a disaggregated rack. It observes all traffic, from control flow to memory accesses. This provides a unique advantage of monitoring the state of the system based on the observed network traffic. This may seem in violation of the end-to-end argument, but we are only proposing to move the management functionality that would perform better at the switch [37]. Taking a decentralized approach to fate-sharing and memory protection requires coordination of all resources in the system, thus incurring an overhead of communication. Moving that functionality to a passive centralized solution which does not require extra hardware should improve system performance.

Chapter 3

Background and assumptions

3.1 Disaggregation

There are two types of disaggregation: full and partial. In full disaggregation each resource is completely independent and attached to the network. For example, a CPU will not have any on board or directly connected RAM. Partial disaggregation is where a CPU will have a small amount of RAM directly attached to it, this RAM acts as an extra cache level and RAM has a small CPU attached to it which acts as a memory controller [16]. Research has been trending towards disaggregating hardware components, with partial disaggregation being the first step [16, 28, 29].

Although disaggregation aims to be deployed at the datacenter scale, there are practical limitations to this, such as interconnect speed and distance between components. Recent research and prototypes tend to focus on disaggregation on a rack-scale for these reasons [1, 10, 16]. Based on the current directions in disaggregation research, we focus our solution on a partially disaggregated rack.

Disaggregation is not only a trend in datacenter architecture, it is very quickly becoming a reality. It has been shown that legacy applications can perform in a partially disaggregated rack-scale environment if bandwidth is greater than 40 Gbps and latency is less than 5 μs [16]. Therefore, we can not only prototype disaggregated racks [1, 6] but also run legacy applications on them with commodity interconnects.

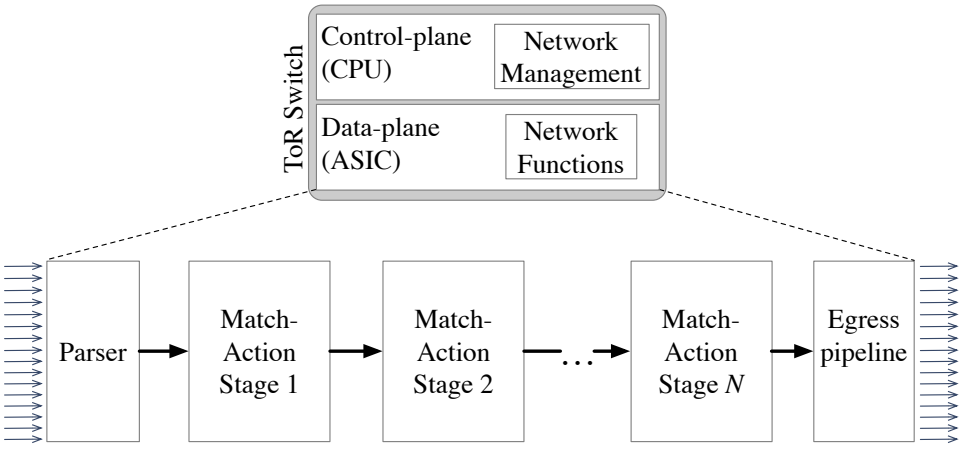

Figure 3.1: Protocol independent switch architecture.

3.2 Programmable switches

Programmable switches provide flexibility to network operators by allowing fast prototyping of new protocols. There are two primary architectures for these switches, we focus on building for protocol independent switch architecture (PISA). PISA follows a pipeline match action architecture in the data-plane of the switch (Figure 3.1). Packets get parsed at ingress by a custom parsing program. They then flow through the each stage's match-action table until they reach the egress queues. If a packet requires special processing, it gets trapped to the control-plane of the switch, which has a control program running on a CPU. This has become the predominant architecture for Barefoot switches [4]. There are two fundamental limitations for programming these switches: 1) the maximum number of pipeline stages is fixed and 2) memory is explicitly tied to a stage and cannot be used by another stage [13]. Neither of these limitations effect our work, as we do not require much memory per stage and we do not require many stages of computation.

3.3 Remote procedure calls

RPCS aim to abstract away the complexities of distributed communication by performing server lookup, marshaling/unmarshaling of data, and network setup for the developer. Although this is an attractive and simplifying abstraction, there are

many challenges with implementing it. We address two in this work: precise failure semantics and pass by reference arguments.

Precise failure semantics. In the ideal case, RPCS would transparently provide local procedure call semantics (*exactly-once*). This is largely debated as being impractical, therefore the semantics are relaxed to *last one* [30]. *Last-one* mean the procedure is called continuously, only the last call will successfully complete [30]. The key to transparent fail-over with last-once semantics lies in orphaned callee discovery and extermination [30]. This requires the control flow state of the program to be maintained and used upon failure.

Modern RPC implementations do not provide any fault recovery and leave it up to the application using RPCS to handle any errors [2, 3, 5]. They tend to focus on providing fast marshaling and unmarshaling of complex data types [3, 5].

Pass by reference arguments. The simple solution, dereferencing the pointer and sending its data in the RPC, is only viable for the pointers to values. Nested pointers (i.e., a struct with a pointer to a pointer) require special consideration [30]. Often, the overhead to implementing such solutions outweighs the benefit of allowing arbitrary pointer arguments. Consequently, most RPC implementations focus on providing multiple language interfaces instead of pass-by-reference arguments [3, 5].

3.4 Distributed shared memory

Distributed shared memory (DSM) systems provide a global address space for applications. DSMS make many design decisions regarding granularity of memory access (i.e., pages or objects), coherence semantics, scalability, and heterogeneity [33]. The different coherence semantics range from release consistency to strict consistency [19, 24, 31, 34]. For example, Grappa provides global linearizable guarantees for their memory model, a strict consistency model [31]. They guarantee that every modification to the data occurs in a serialized manner and every read returns the most recently written value. Whereas, TreadMarks provides release consistency, which allows processors to delay exposing changes to shared data until a synchronization access occurs (acquire or release) [19]. This allowed TreadMarks to improve upon their performance because they did not require heavy

synchronization overhead on all processors and implement a lazy version of release consistency.

Chapter 4

Bifröst Semantics

4.1 Memory semantics

Bifröst provides strict memory consistency semantics, where the most recently written value will be read by the processor. Concurrent writes by the same node will be serialized and processed in order at the memory server. This is enforced through a notion of *control* and *lease*. When a CPU allocates a region of memory, it automatically assumes control over that region. When a pointer to a region of memory is passed as an argument, the memory is implicitly leased to the receiving CPU.

Figure 4.1 displays the memory semantics during an RPC. The arguments (`ptr1`) are first alloc'd by `A` (caller). This can be done well before the call (i.e., read all data into memory) or be done on a per call basis. `ptr1` are then passed in an RPC, leasing them to the callee, the caller maintains control over the memory but can only read it while the callee maintains the lease (green section). Control transfer occurs only on returning a result to the caller. First `B` (callee) allocates memory for `ptr2` and has control over the memory (blue region). When `B` returns `ptr2`, it transfers this control to `A`, converting the access from blue to orange. This allows the caller to decide which memory should be freed (`ptr1` or `ptr2`).

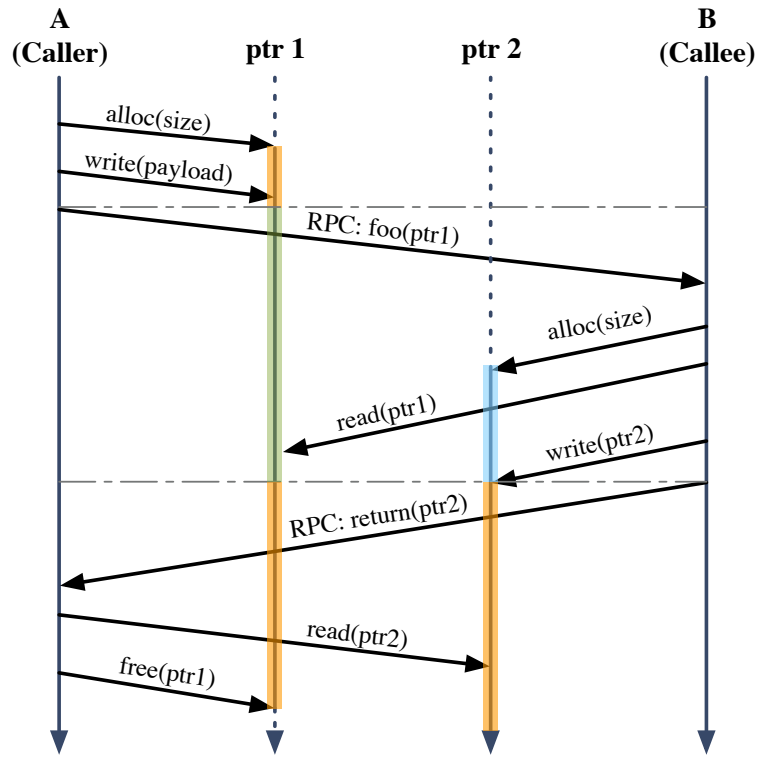

Figure 4.1: Bifröst memory semantics during a RPC. The orange region represents A's over the pointer. Blue represents B's control over a pointer. Green represents a leased pointer, therefore read-only access.

4.2 Call semantics

Under all conditions, Bifröst provides *exactly-once* semantics. We define exactly-once semantics as both the caller and the callee observe the call executing exactly once. This is a harder requirement than current RPC systems, as they allow for retransmissions, whereas we consider those to be more than one call (giving *last-one* semantics). To illustrate this, we focus on a use case where the caller and callee use pass-by-reference for both argument and result.

Normal operations. The caller makes a RPC with the desired method and arguments, which is routed to the appropriate callee (Figure 1.1(DDC RPC) dotted green arrow). The callee then reads in the global address argument (Figure 1.1(DDC

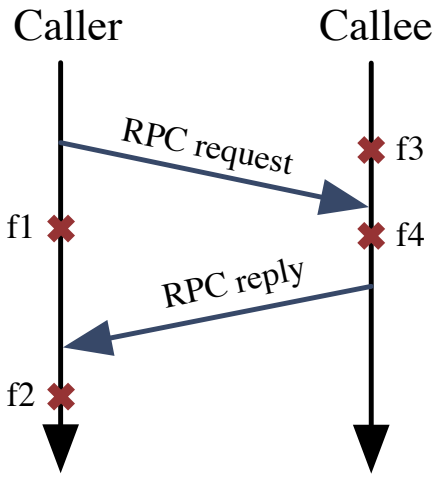

Figure 4.2: Call semantics with possible failure points.

RPC) blue arrow), if any, and performs the procedure. Once the callee is finished, it writes back any relevant output (Figure 1.1(DDC RPC) red arrow) and returns the result address. The caller then reads the data and continues. Here, Bifröst maintains exactly-once semantics, but the challenge arises under failure conditions.

Caller failure. If the caller fails at point `f1` in Figure 4.2, then the caller and callee consider the RPC executed, but there is no caller to return to. If the caller fails at point `f2` in Figure 4.2, both the caller and the callee consider the RPC executed once. But, when the caller is restarted, it would have lost this state. There are two options: fate-share and recovery. In fate-sharing the callee is forced to fail if the caller fails, and both are restarted. This provides exactly-once semantics as both the caller and callee do not know of the previous attempted execution after reboot.

Exactly-once semantics can be achieved using RPC-based checkpointing. The caller considers the RPC initiated when it successfully sends the request. At this point, it waits for the callee's response. A checkpoint is taken shortly before the RPC has been initiated. When the caller fails, it is restarted with this checkpoint, thus preserving the state immediately after the RPC has been sent. Both the caller and the callee both have a record of the RPC executing only once.

Callee failure. If the callee fails at point `f3` in Figure 4.2, then the callee is not

alive to field the caller's request. This can be fixed by retransmitting the request on the caller side until the callee is back, but this violates the definition of exactly-once. If the callee fails at `f4` in Figure 4.2, then both the caller and callee consider the RPC initiated but not executed. The first failure can be addressed by in-network retransmission of the packet once the callee has been rebooted. Essentially, the caller does not retransmit the message, from its perspective the RPC was only called once, but the underlying network handles retransmission of the packet to the callee.

The failure case at `f4` can be handled using the same RPC checkpointing scheme described above. Here, the callee is checkpointed immediately after the RPC is received. Therefore it can restart right at the point where it considers the RPC initiated but not executed. This poses problems for non-idempotent operations on global memory. To handle non-idempotent operations, the global memory used by the callee will fate-share with the callee. Thus requiring the callee, upon reboot, to reload the arguments and any other state from the global memory snapshot.

Chapter 5

Bifröst design

Figure 5.1 shows an overview of the Bifröst architecture. Each of the compute resources that comprise a single machine are connected via the network to the global address space. The global address space is provided by a distributed shared memory system run on top of the memory resources.

Application processes call RPCS using the Bifröst RPC library. This RPC library uses the Bifröst application programming interface (API) to manage global memory. The semantics described in Section 4 are enforced in the ToR switch by the control-plane program *Gatekeeper*.

5.1 API

Bifröst presents two APIS, one for accessing global memory and the other for RPCS (Table 5.1). The memory API provides four basic primitives for operating on global memory: alloc, read, write, and free. All of these operations are performed on global address pointers. The RPC API only provides two calls, one for the caller and one for the callee. The RPC library handles all marshaling and unmarshaling of the arguments and results and handles the translation of a method call to a remote call.

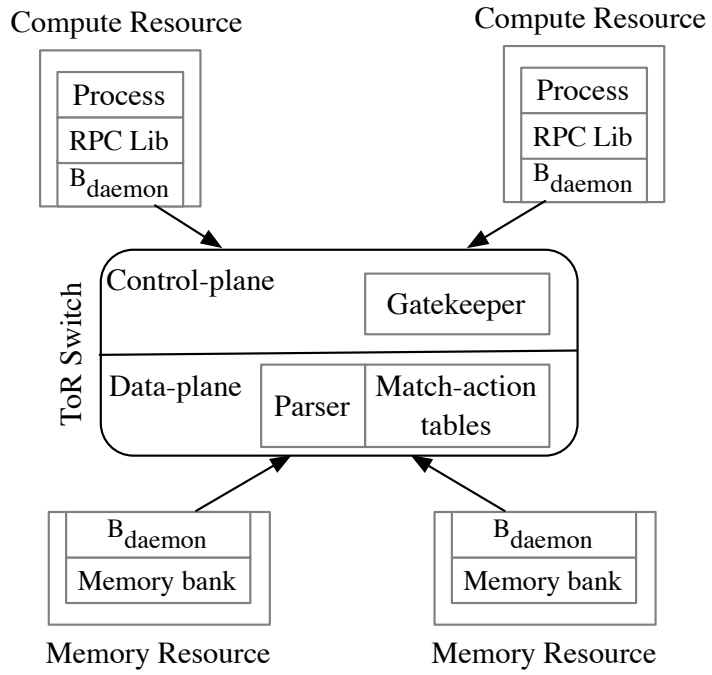

Figure 5.1: Overview of Bifröst architecture.

	API Call
Memory	ptr← alloc(int size)
	int← read(char* buf, int len, ptr addr)
	int← write(char* buf, int len, ptr addr)
	free(ptr addr)
RPC caller	result←<method_name>(args, ...)
RPC callee	export(<method_name>)

Table 5.1: Bifröst API.

5.2 Network protocol

Figure 5.2 shows the Bifröst packet header. It contains a Bifröst identifier that the parser on the switch data-plane uses to determine if the packet should go through the Bifröst parsing tree. The call type has four possible values: CALL (1), REPLY (2), EXCEPTION (3), and ONEWAY (4). This is drawn from Thrift RPC implementation. Next is the length of the method name and then the method name

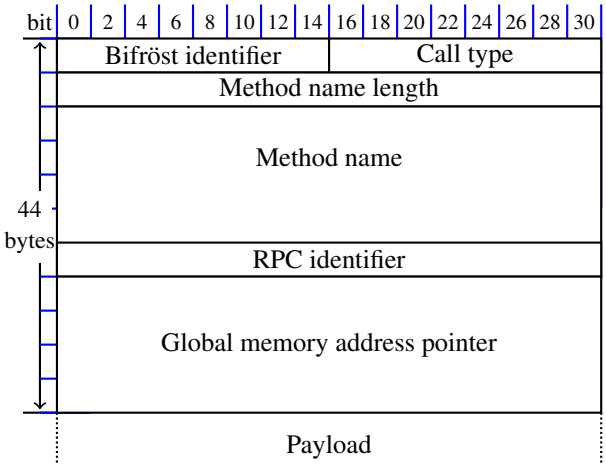

Figure 5.2: Bifröst packet header.

itself. We fix the method name to be 16 bytes to avoid variable length parsing in the switch. The RPC identifier is a unique number which denotes a particular RPC operation for this caller.

The global memory address pointer is a 16 byte address which points to the pass by reference arguments. When multiple arguments are pass by reference, the RPC library allocates memory for each argument, then writes the data in flattened form to one global address. This global address is then sent in the RPC. This removes the need to parse a variable list of pointer arguments to enforce the memory semantics discussed in Section 4.1. The rest of the payload is not parsed by the switch.

5.3 Gatekeeper: switch control-plane

Gatekeeper, run on the control-plane of the switch, tracks and manages all Bifröst traffic. Gatekeeper performs three main tasks: resource management, memory protection, and fate-sharing and fault tolerance for RPCS. It enforces every decision as a match-action rule in the data-plane of the switch.

Resource management. To present the user with a single machine, the controlling entity must be able to determine available resources. When a resource becomes alive, either from reboot or initial plug in, it automatically sends an initiation message. This message contains resource type and capacity or specification.

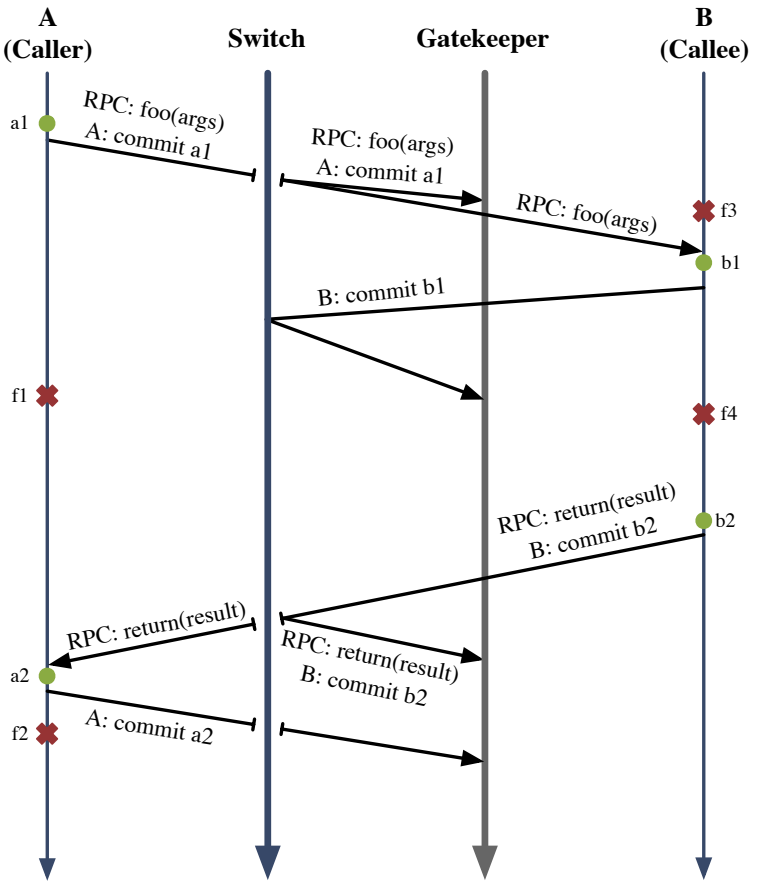

Figure 5.3: Bifröst RPC fault recovery scheme.

The Gatekeeper maintains a list of these resources and their status (free, in-use, or failed). When a program has been compiled, procedures are then assigned to compute resources. The compute resources send a similar initialization message to the switch. This message is trapped to the control-plane where Gatekeeper generates a match-action rule to automatically forwarded any Bifröst packet to that particular compute resource for processing. This alleviates the need to directly connect RPC callers and callees, it provides the RPC registration service in the switch data-plane.

Memory protection. The switch data-plane drops all packets for a pointer by default. It only allows access if a table rule is generated to allow access. Based on the memory semantics described earlier (Section 4.1), when the data-plane parses

a Bifröst RPC packet with a global address pointer, it traps to the Gatekeeper in the control-plane. Gatekeeper generates a match-action rule to allow the destination IP address (callee) to read that pointer. It removes the rule which allows the source IP address (caller) to write to that pointer. These rules match based on a pointer, operation, and source IP address from a memory access packet.

The access control table update is represented by the first dashed line and the change from orange access to green access in Figure 4.1. When the RPC completes, the data-plane parses the response RPC packet, if the reply packet contains a global address, it traps to Gatekeeper which removes the rule allowing the source IP address (callee) to access the pointer. Gatekeeper also must update the table to allow the destination IP address (caller) to perform any memory operation on the returned pointer and the arguments previously sent.

When the data-plane parses any memory access packet, it looks at the operation, the source address, and the pointer address. If the operation is allocation, it traps to Gatekeeper which generates a table rule to allow all memory access packets from the requesting address. If the operation is a free, Gatekeeper removes any match-action rule regarding that pointer. It does not update the tables for a read or write operation.

Fate-sharing and fault tolerance. The switch data-plane traps to the control-plane when it encounters an RPC packet, here Gatekeeper maintains a graph of active RPCS. This graph represents the control and data flow of the program. It allows Gatekeeper to not only track nested RPCS but also pointers that are passed in multiple RPCS. When a resource fails, Gatekeeper determines the failure domain of that RPC and if it should be recovered. These decisions can be made on a call by call basis, it is also possible to make them programmable by the developer [10]. Once the failure domain is computed, the control-plane removes rules in the match-action tables which forwarded packets to the blacklisted IP addresses. By default, these packets will be dropped.

We deploy checkpoint and rollback recovery for both caller and callee failures. We base our checkpoints and committal of checkpoints on RPC calls (Figure 5.3). This means we have a guarantee of RPC state at each particular checkpoint. Gatekeeper keeps track of the most recent checkpoint committed by a node.

There are four points in which a checkpoint is created and committed: RPC

initiation (a1), RPC initiation received (b1), RPC reply (b2), RPC reply received (a2). When A (caller) creates the checkpoint a1, the checkpoint is committed to Gatekeeper when the RPC sent to B (callee). This ensures that the checkpoint a1 is only committed when the RPC is actually sent across the network. At this point, we guarantee that A views the RPC as "called".

When the RPC is received by B, B immediately checkpoints its state (b1) and commits that to Gatekeeper. Once b1 is committed, B can proceed with computing the procedure. This ensures that B will restart at the beginning of computation, ensuring the RPC is received once, but never executed more than once (based on the view of the callee). When B has completed the procedure, it checkpoints its state again (b2) and commits it with the return of the RPC. Thus representing that B considers the RPC "completed". When A has received the RPC reply, it immediately checkpoints its state (a2) and commits it before continuing computation. Once a2 is committed, A considers the RPC "completed".

Each checkpoint represents the RPC status to the particular node making the checkpoint. This aids in recovery as we can determine whether or not the caller thinks it called the procedure, the callee received the request, the callee completed the request, and if the caller received the result. Maintaining exactly-once semantics is still not trivial, especially in the cases where the caller and callee checkpoints do not reflect the same status, in particular, f4 in Figure 5.3. To solve this, we have Gatekeeper maintain a most recent RPC for every caller/callee pair. When f4 occurs, Gatekeeper will see that A considers the RPC called, but B failed before receiving that call. Therefore, Gatekeeper will restart B from the most recent checkpoint and then replay the last RPC from the pair to synchronize their view of the RPC. A does not know of the replayed packet and still considers the RPC to be called once.

5.4 Bifröst daemons

Bifröst requires coordination on the resource side to achieve transparency to the application, execute checkpointing or snapshotting, and perform memory clean-up. The Bifröst daemons change roles depending on which type of resource they run on.

Compute daemons keep track of the number of times a pointer has been leased and initiate the synchronous checkpoint before an RPC is sent across the network and before the RPC return is passed up to the process. When performing the checkpointing, the compute daemon appends the commit information on the end of the Bifröst packet. This extra information is removed by the compute daemon on the callee node. Memory daemons handle memory API requests, perform memory snapshotting on a specified region, and service memory clean-up requests.

Chapter 6

Implementation

To prototype our design, we modify an existing RPC framework (Thrift), and built a basic DSM system. The switch data-plane program is written in P4 and Gatekeeper is written in C++. We simulate the network topology in Mininet with a P4 switch.

6.1 Thrift modifications

Thrift is an RPC library originally developed at Facebook, but open-sourced as an Apache project [3]. Thrift provides flexibility with different abstraction layers: thrift file, `TClient`/`TProcessor`, `TProtocol`, `TTransport` (buffered, framed, etc.), and `TSocket`. The user specifies the thrift file which is then compiled and generates the `TClient`/`TProcessor` for both the client and the server respectively. `TProtocol` defines the marshaling and unmarshaling for every Thrift data type. When a data type is marshaled it is written to the transport. For complex data types, such as lists Thrift marshals each item in the list. `TTransport` defines wrapper functions to perform network operations. `TSocket` performs the actual network I/O functions.

We start with the Thrift `c_glib` library, using the binary protocol (sends the data as raw bytes) and the buffered transport (buffers the data before calling socket send or receive). This means, when Thrift marshals a data type, it "writes" to the buffered transport. If the write buffer is full, the transport sends the message over TCP, if the buffer isn't full, it writes the message to the buffer. The same is true for

reads. This can be inefficient when the buffer size is much smaller than the data being sent.

Bifröst requires UDP to perform the rerouting and packet drops required to enforce our desired semantics. The entire Thrift design is based on the connection abstraction of TCP. This required us to modify the `TTransport` layer. We created a buffered transport for UDP which uses a UDP socket. The UDP socket performs whole reads of messages, which are then buffered at the UDP buffered transport layer. We also created a connectionless server using the UDP socket. Instead of listening and accepting connections, the server receives an RPC, processes it, and replies. This removes the need for any handshake between the client and server as well as reduces the number of open sockets. It currently does not handle multiple connections, nor does it queue outstanding requests. We plan on addressing that in future work.

6.2 DSM system

Our basic DSM system exposes a key value store interface, where the global address is the key and the data is the value. We chose to embed the global address and operation type in an IPv6 address to aid in load balancing, memory migration, and, it is addressable from any requesting machine. The first four bytes of the IPv6 address are zero, the fifth byte is the DSM system prefix. The sixth byte is the DSM server machine ID. The seventh byte is the operation (allocate, read, write, free). The eighth byte is the arguments (if any). Finally, the last eight bytes are the 64 bit pointer. Embedding the global address in IPv6 is not required for our system. The global address can be passed in an application level header or payload and parsed at the server side.

6.3 P4 data-plane program

P4 is a highly reconfigurable, protocol and target (i.e., switch) independent switch data-plane language [9]. P4 programs for the PISA we described in Section 3. A P4 program has the following components: ingress parser, match-action tables, actions, and header definitions.

The packets flow along the pipeline by first being parsed based on the parsing

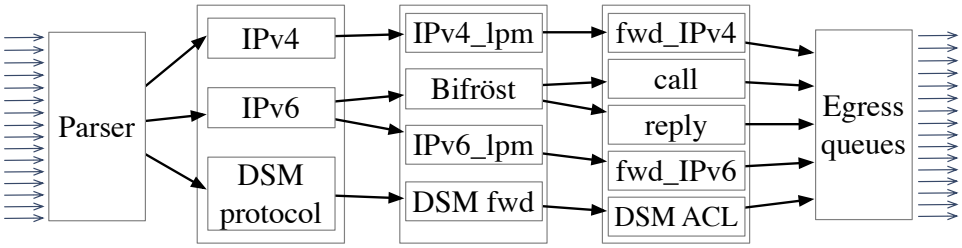

Figure 6.1: Bifröst P4 pipeline.

rules (Figure 6.1). Then our P4 program checks to determine if the packet is a Bifröst packet based on the parsed header fields. This allows it to co-exist with other network functions on the switch. Once it determines it is a Bifröst packet, it applies the Bifröst match-action rules. When the switch data-plane encounters a Bifröst call and reply or a DSM allocate, it traps to the control-plane CPU, where Gatekeeper runs, by generating a `digest`.

6.4 Gatekeeper

Gatekeeper is written in C++ and runs on the switch control-plane CPU. When Gatekeeper receives a trap from the switch data-plane it determines which type event has occurred: `RPC_initiation`, `RPC_return`, `RPC_failure`, `Mem_alloc`, `Mem_free`. In the case of `RPC_initiation`, Gatekeeper parses the packet to determine if any global address is used. It then generates match-action rules to enforce memory protection updates on the global addresses involved in the RPC call. Finally, it creates an RPC entry in its control flow graph. The control flow graph is maintained in a node-centric data structure. Each node represents a caller or callee with a directed edge between the two. Each directed edge contains the call method name and pointer argument. When a node is added to the graph, it is also added to a list of current RPCS. This list contains node pointers into the graph.

When an RPC returns (`RPC_return`, Gatekeeper must perform clean up operations on its graph. First, it updates the match-action rules for the global addresses involved in the RPC. Then it removes the callee and caller pair from the graph (assuming the caller does not have other outstanding RPCS).

When Gatekeeper receives an `RPC_failure`, it walks the graph to see what

caller or callees the failed processor was associated with. Gatekeeper then builds the failure domain based off of those associations. It will initiate memory clean up for the pointers controlled by the failed processor IP address. Simultaneously, it will get a new processor and initiate it from the checkpoint. Any requests going to the failed processor are queued at Gatekeeper. Once the new processor is initialized, the held requests are forwarded to the new processor. Any entry in the tables that forwarded to the failed processor are re-written to point to the new processor.

For `Mem_alloc` and `Mem_free`, Gatekeeper updates the access control list for the memory pointers. It adds an entry for `Mem_alloc` and removes an entry for `Mem_free`, assuming the requesting IP address is the one that controls that memory pointer. If the requesting IP address does not control the pointer, the packet is dropped.

Chapter 7

Evaluation

We focus our evaluation on the overhead of our system compared to the overhead of Thrift. We attempt to answer five questions regarding our system:

- How realistic is our test environment?
- What is the base overhead of Thrift and Bifröst?
- What is the impact of our Thrift modifications?
- What is the cost of Thrift and Bifröst on a simple workload?
- What is the latency breakdown in Thrift and Bifröst?

7.1 Methodology

We perform our tests using a network simulation environment called Mininet [23]. Mininet simulates a customized network topology, allowing the user to rapidly prototype and test switch and application code. It is easily customizable, we use it with a P4 switch to prototype our P4 program and Gatekeeper. We also use it with an OpenV Switch (OVS) for our performance testing.

Our topology consists of six machines: one RPC client (`c1`), two RPC servers (`s1`, `s2`), and three DSM servers (`m1`, `m2`, `m3`). `s1` and `s2` run two different RPC services. `s1` handles ping requests, `s2` handles echo and array operation requests. `m1`, `m2`, and `m3` run the memory daemon to service memory access requests.

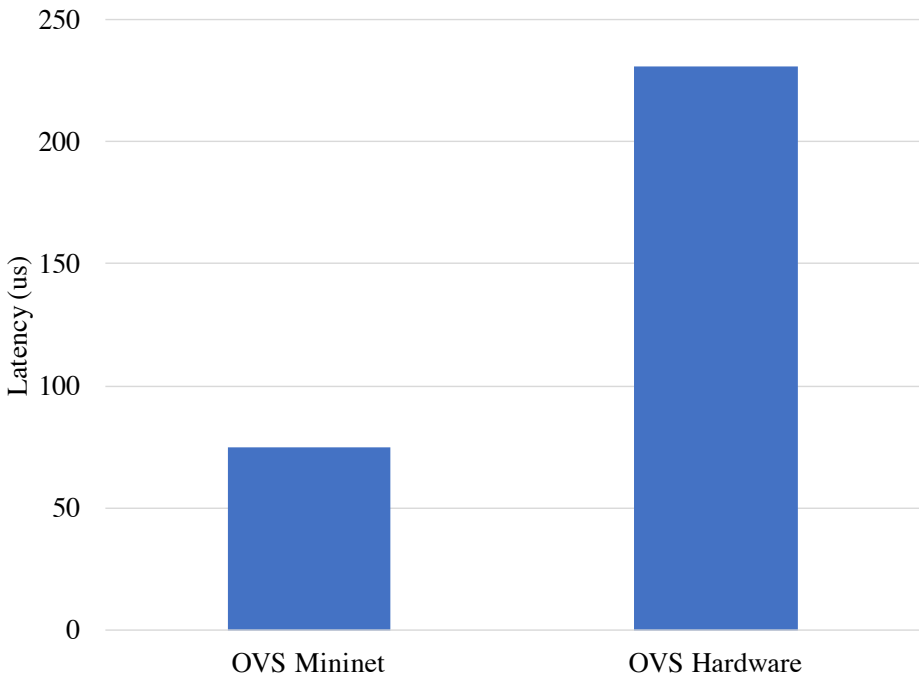

Figure 7.1: Latency comparison using `ping6` on OVS Mininet and real OVS cluster, averaged over 1000 pings.

We use two measurement programs for our baseline measurements: `ping6` and `netperf`. `ping6` uses Internet control message protocol (ICMP) to request and receive an echo between two nodes and measures the round trip time (RTT) in milliseconds. `netperf` performs network testing between two hosts, measuring a variety of metrics such as bandwidth and RTT. We use two specific `netperf` tests: `TCP_RR` and `UDP_RR`. `TCP_RR` stands for TCP request/receive. It performs as many requests with a specified payload in ten seconds. It then outputs the 50th percentile, 90th percentile, 99th percentile, mean, and standard deviation latency measurements in μs. `UDP_RR` performs the same test, except over an UDP connection.

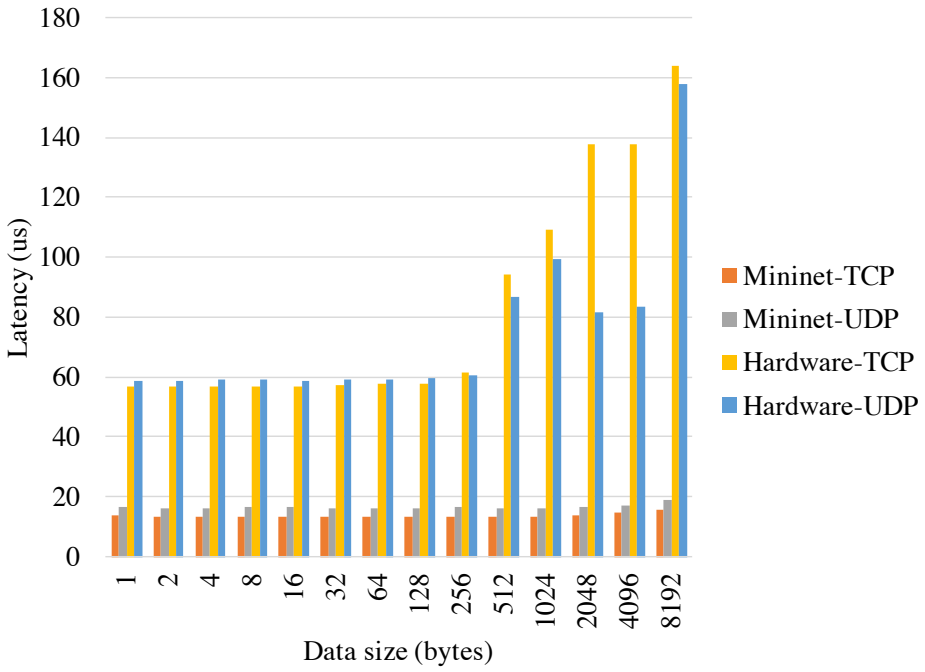

Figure 7.2: Latency comparison using `netperf` with TCP and UDP on OVS Mininet and real OVS cluster. Averaged over 1000 pings and with varying payload size.

7.2 How realistic is our test environment?

We look at the cost of pings (using `ping6` and `netperf`) in Mininet compared to two servers with 10Gb NICs connected with an OVS. Figure 7.1 shows that the ping latency of Mininet is lower than hardware. This is expected as Mininet runs on a single host and does not require traversing the physical NIC. Mininet has an average RTT of 75 μs whereas hardware has an average RTT of 231 μs. When looking at protocol specific numbers, Mininet outperforms the servers by a factor 4x for TCP and 3x for UDP on small data sizes. With large data sizes, Mininet starts to outperform the real servers by a factor of 10x for TCP and 7x for UDP, as expected (Figure 7.2).

Based on these measurements, Mininet shows an optimistic performance compared to real hardware. We, therefore, base our performance evaluation on relative

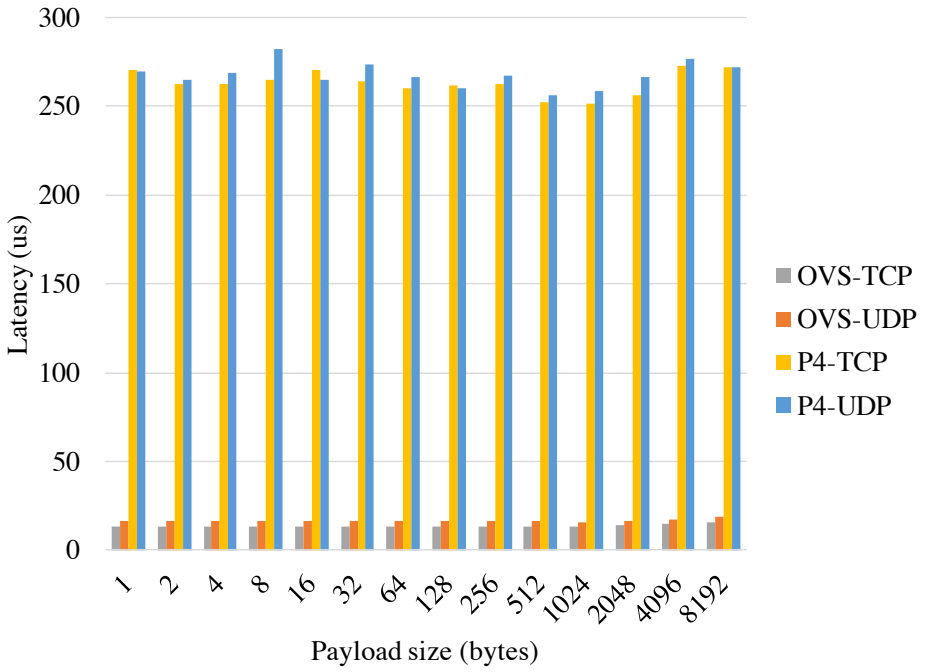

Figure 7.3: Baseline latency in μs for OVS Mininet compared to P4 Mininet with ranging payload size. Measurements were taken with `netperf`.

overhead.

We also ran `netperf` on the OVS and P4 versions of Mininet with varying payload sizes (Figure 7.3). The P4 switch in Mininet is an average 20x slower for TCP and 16x for UDP. We believe this is due to the P4 parsing overhead and that the P4 switch implementation is not optimized. We plan to investigate this further in future work. But, because of the large performance overhead, we use the OVS Mininet for the rest of our tests.

7.3 What is the base overhead of Thrift and Bifröst?

To determine the base overhead of our system, we run two microbenchmarks: ping test and echo test. The ping test calls a "ping" RPC, which the server just returns ACK. This is the simplest RPC, with no parameters or return value. The total UDP/TCP payload size is 29 bytes, including the Thrift header. To provide a baseline,

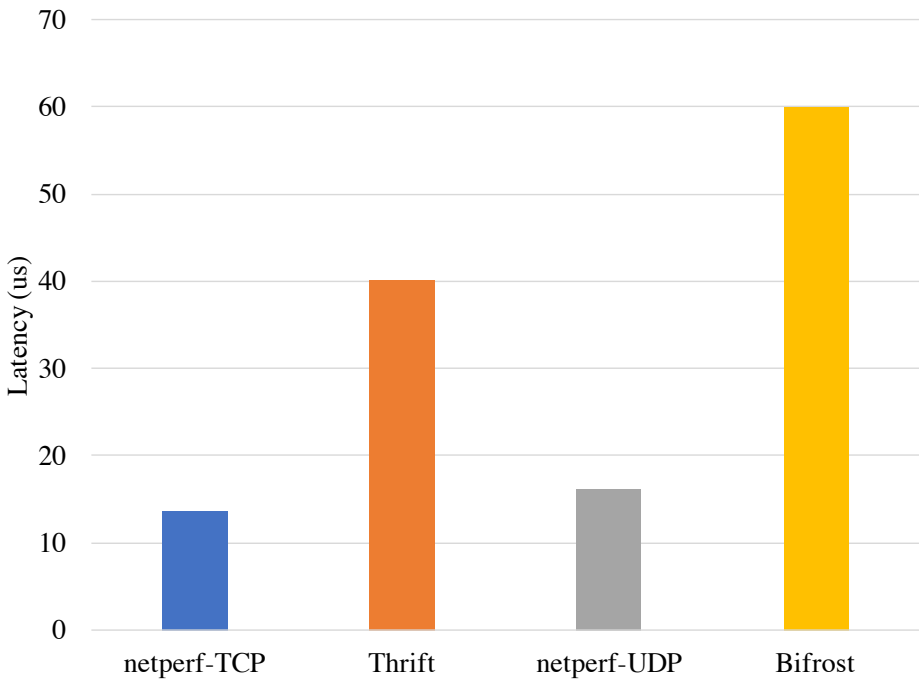

Figure 7.4: Thrift and Bifröst running on OVS Mininet compared to the `netperf` round trip latency measurement for TCP and UDP.

we also ran `netperf` with a 29 byte request payload and a 1 byte reply payload. We ran the ping test 100 times and took the average latency. Thrift has a 3.09x overhead compared to TCP on the OVS Mininet and Bifröst has a 3.78x overhead compared to UDP on the OVS Mininet (Figure 7.4). Although the Bifröst overhead is slightly higher than the Thrift overhead (only 0.69 difference), we believe this difference is negligible.

7.4 What is the impact of UDP vs TCP?

To determine how the difference of protocol (UDP vs. TCP) effects our performance measurements of Bifröst, we compare the performance of regular Thrift (over TCP) with our Thrift UDP implementation, but no Bifröst management or DSM system. We ran a the ping test on the OVS switch for 100 iterations and averaged the RTT in μs. Thrift over TCP had an average latency of 37 μs, whereas

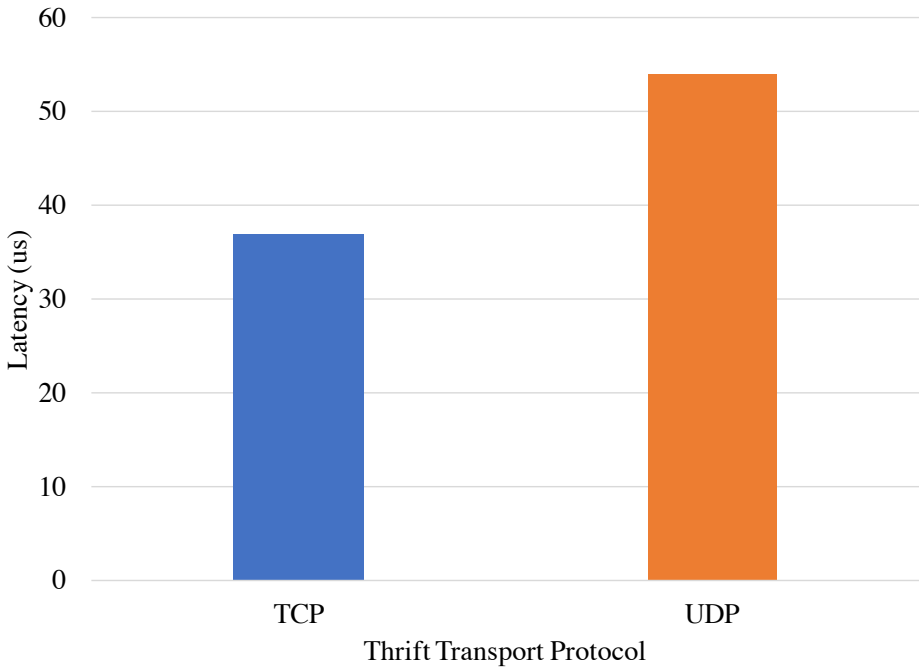

Figure 7.5: Thrift RPC ping test using TCP and UDP as underlying transports, averaged over 100 iterations.

Thrift over UDP had an average latency of 54 μs (Figure 7.5). The fact that UDP increases the latency by ~45% is interesting, as UDP should have better performance than TCP. We plan to investigate why Thrift UDP performs worse in future work.

7.5 What are the overheads on a simple workload?

To get a preliminary idea of what the performance of Bifröst would be running real-world workloads, we tested two simple workloads: increment array and add arrays. Increment array calls an RPC with a byte array and a byte as parameters and expects a byte array of the same length as a return value. The server receives this request and increments the passed array with the passed value. Add arrays sends two byte arrays and expects a byte array of the same length as a return value. The server performs the element-wise addition of the two arrays.

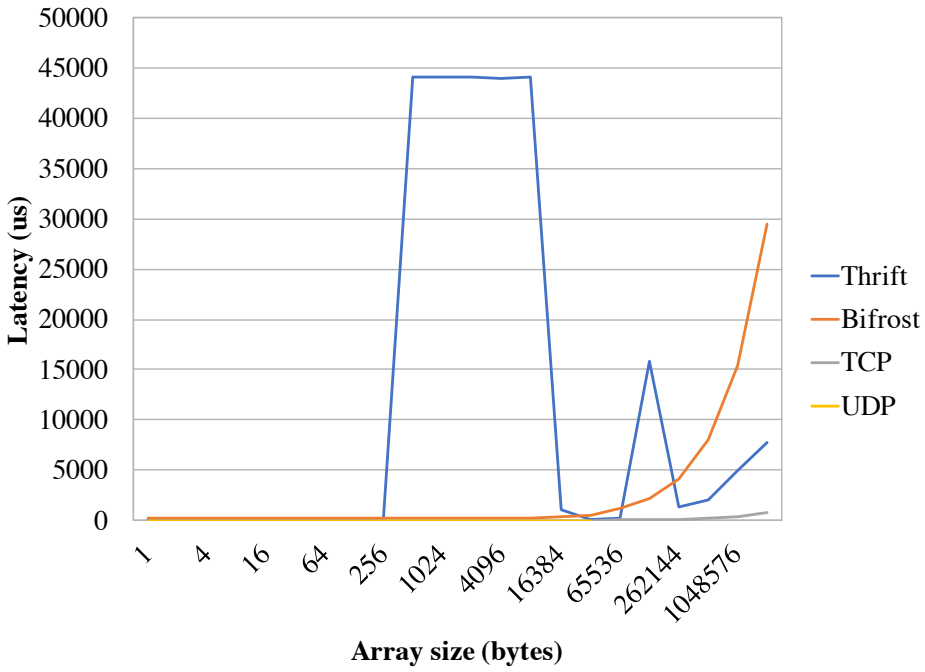

Figure 7.6: Increment array RPC test on OVS Mininet with Thrift, Bifröst, TCP `netperf` baseline and UDP `netperf` baseline. Averaged over 100 iterations and varying data size.

We ran the test on varying payload sizes (up to 2 MB) and for 100 iterations. Figure 7.6 shows the average latency of Bifröst, Thrift, UDP baseline, and TCP baseline on the OVS Mininet. Bifröst's performance increases exponentially with the data size. Thrift's performance also increases, but does so slightly more erratically. Compared to their baselines, Bifröst has, on average, a 28x overhead compared to UDP and Thrift has a 21x average overhead compared to TCP. This is due to the extra reads and writes Bifröst must perform to access global memory.

Figure 7.7 shows the comparison between Bifröst, Thrift, UDP, and TCP over the OVS Mininet running the add arrays workload. Bifröst performs worse than Thrift on average. Bifröst has an overhead of 41x compared to UDP and Thrift has an overhead of 22x.

We found that Thrift had very odd behavior once the packet is larger than 512 bytes, then subsides when the packet is larger than 9000 bytes. It seems that the

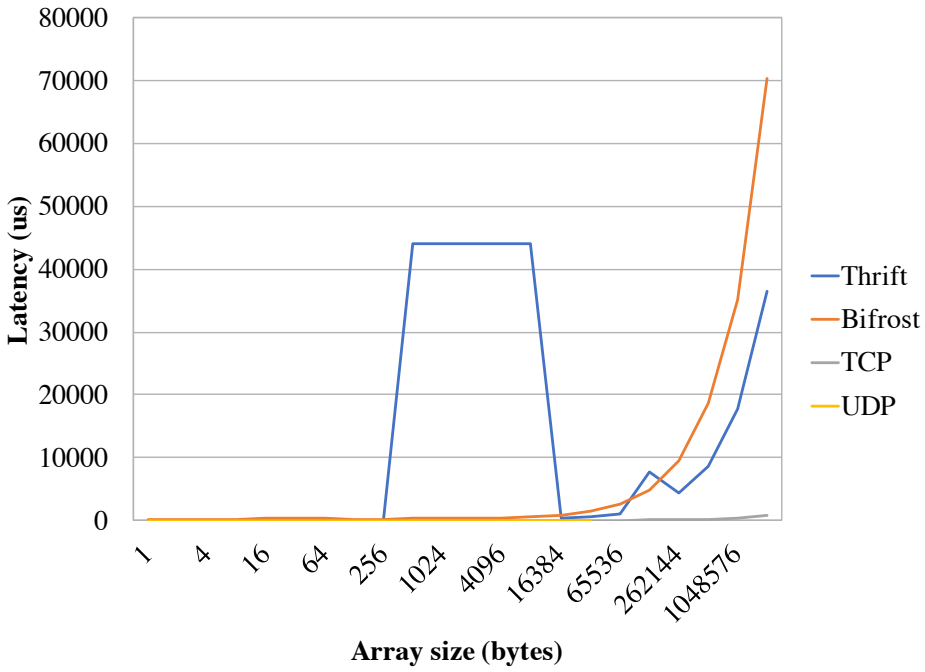

Figure 7.7: Add array RPC test on OVS Mininet with Thrift, Bifröst, TCP baseline and UDP baseline. The baselines are NetPerf latencies. Averaged over 100 iterations and varying data size.

TCP segment size gets stuck at 512 bytes, even though the packet is larger, which causes malformed packets. The negotiated maximum segment size (MSS) is 8940, which is our maximum transmission unit (MTU), 9000 bytes, minus the TCP and IP headers. We found the same behavior when running on the P4 Mininet. There is also a secondary spike which occurs at 131072 bytes. We plan to investigate both spikes in future work.

7.6 What is the latency breakdown in Thrift and Bifröst?

To discover where most time is spent, we also gather breakdown measurements for both Thrift and Bifröst over 100 iterations of each RPC and an array size of 16384 bytes. This will help pinpoint possible bottlenecks in both systems. We measured the costs into pre-processing, marshaling, network, unmarshaling, server

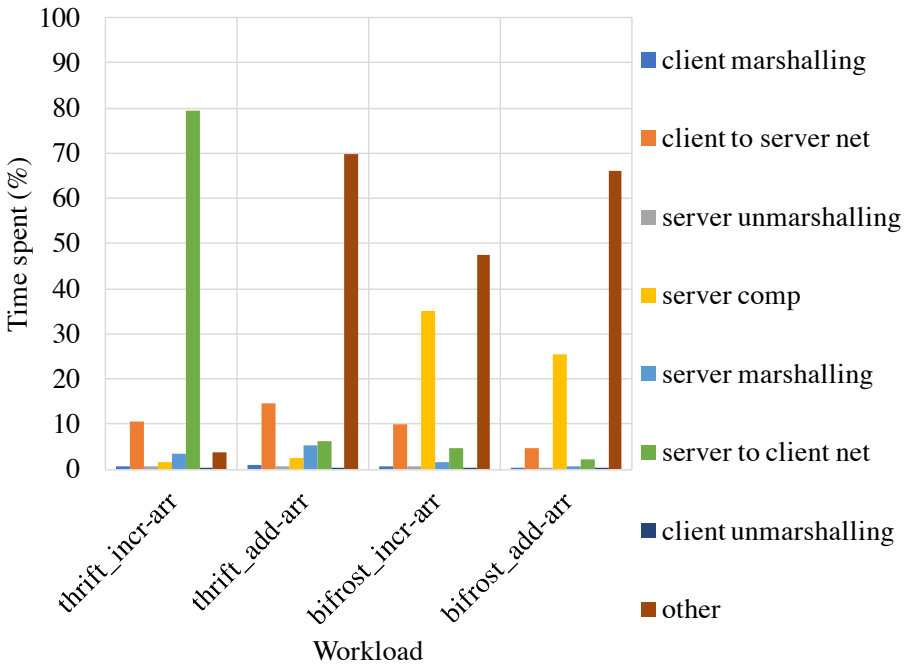

Figure 7.8: Breakdown of where time is spent for each small workload and a data size of 16384 bytes, averaged over 100 iterations.

computation, and post-processing. Pre- and post-processing occur on the client side, this is where the arrays are populated, written or read from remote memory or copied. Marshaling and unmarshaling occurs on both the client and server side.

Figures 7.8 shows a breakdown of our results. Bifröst spent the most time during the pre- and post-processing stages (labeled as other in Figure 7.8). Outside of that, Bifröst spent the most time in server computation. Both these are expected as the memory access calls occur during pre-processing, post-processing, and server computation in Bifröst.

Thrift spent most of the time in the processing stages for add array. Whereas Thrift spent most of the increment array latency sending from the server to the client. We found these results interesting, as we expected the majority of Thrift latency to be in the network. This is only true in the increment array test, where 80% of the time is the server responding to the client. It is interesting that these

times are not reflective of their payload size either, as increment array is returning less data than it sent. Therefore, it would make sense that the client to server network time would be the largest in Thrift, not the server to the client. We plan to investigate this with the latency spike we see in TCP (Figure 7.6 and Figure 7.7).

Chapter 8

Discussion

8.1 Limitations

There are several limitations to our current design based on the assumptions and trade-offs we made. Our design currently assumes rack-scale disaggregation. This is currently the most plausible form of disaggregation, but limits the scalability of our solution. Since we rely on the switch to have a global view of the resources, our design does not directly translate to a scale larger than a rack. It is not impossible to scale our design beyond a rack. It requires more complexity and coordination, as now Gatekeeper must act as a distributed system which makes decisions about failures and memory across racks. This can introduce many complexities, such as rack failures, switch failures, and distributed consensus.

Another limitation of our system is that we do not handle any network failures. In particular, we consider the network to be lossless, which is not true for some commodity networks. To address this, we would need to add retransmissions and timeouts to our protocol, which in turn, will add a performance overhead.

We also limit our design by having fixed size method names and only one global pointer per RPC. We did this to simplify the parsing on the data-plane. There is a trade-off between variable length header entries and performance of the P4 parsing script. We elected to choose performance of the script over supporting variable-length method names and pointer lists. We could modify the P4 program to handle variable-length parsing, but, do not find it necessary at this point.

8.2 Future work

We split our future work into two categories: system improvement and evaluation improvement. System improvement describes optimizations or functionality we wish to add to our system. Evaluation improvement describes any questions regarding our current evaluation we wish to address or more tests we wish to run.

8.2.1 System improvements

Some of our performance overhead is due to the unmarshaling on the server side. We plan to mitigate this by creating a custom Thrift type for our shared pointer scheme. These types will be used just like C pointers but the RPC library will handle the marshaling and unmarshaling into the IPv6 pointer mechanism our DSM uses. This will be advantageous as the shared pointers are currently stored as Thrift byte arrays. This means Thrift will marshal and unmarshal them byte by byte, allowing for variable byte arrays. Since our pointers are fixed size, we can send a fixed size byte array instead of marshaling in pieces. This will also aid in the transparency of the developer, as they will just be using a different pointer type in C, but all the access semantics remain the same.

As stated in Section 6, our server does not handle multiple requests from clients. We plan to address this by creating a multi-threaded server which maintains a thread pool for servicing requests. As requests come in, it logs the thread ID of the assigned thread then passes the packet. If all threads are busy, it will put the packet in a buffer that is FIFO. Once a thread is completed (i.e., sends the REPLY), it will assign it a packet from the buffer.

We'd also like to implement our fault tolerance design and fate-sharing enforcement. This requires implementing a snapshotting mechanism on the memory daemon and a checkpointing mechanism on the compute daemon. Both of these mechanisms must have low execution time and be synchronous, to ensure no outgoing network packets occur while the snapshot or checkpoint is taken. This atomicity is difficult to achieve, but it will provide a basis for building new fault tolerance techniques.

In this work, we only considered CPU failures, but we must also design and evaluate memory fault tolerance techniques. We hope to show that performing

most of the coordination and computation at the switch will allow for new fault tolerance techniques.

Once the fault tolerance mechanism and fate-sharing enforcement is implemented for both memory and CPU, we would like to experiment with different possible fate-sharing models and fault tolerance techniques as described in [10]. In particular, we believe the tainted fate-sharing model would be advantageous for our DSM system. There might also be advantages to having a specific fate-sharing model for nested RPCS or different fate-sharing models and fault tolerance depending on if the processor was the caller or callee.

8.2.2 Evaluation improvements

Our current evaluation is limited in only showing the performance characteristics of Thrift vs. Bifröst and evaluating the end to end latency of each operation. We hope to expand upon our evaluation in multiple ways, first addressing interesting questions raised from our current numbers and secondly, measuring the effectiveness of other aspects of our system.

Our next step would be to measure the memory protection in the switch, looking at the overhead for the P4 program in the data-plane and Gatekeeper in the control-plane. We also plan to test the memory protection, testing to see if the semantics we describe align with the implementation. For example, the application will attempt to access a memory pointer that it does not control. We then plan to compare our memory protection scheme to another DSM with the same semantics (strict consistency). This will allow us to compare our performance overhead for strict consistency to their implementation.

Once the fault tolerance technique is implemented we will also evaluate it for correctness and performance overhead. We plan to do so in a similar manner to memory protection. First, we will profile it to determine where it spends the most time and to determine the overhead of the P4 program. Then we will test the correctness by injecting faults while an application is running. Finally, we will compare our implementation to an application which provides fault tolerance for RPCS.

Next, when the pointers are integrated into Thrift, we hope to modify a multi-

threaded application to use Bifröst. There we can do a full macrobenchmark of the performance of that application on Bifröst compared to a single machine, showing relative performance overheads. Then we will test for when failures occur, characterizing the failure the application has and measuring the time it takes Bifröst to recover. Finally, we hope to run these tests on real hardware instead of a simulator. This will require us to obtain a programmable switch, at least 40 Gbps Ethernet, and at least 40 Gb NICs.

Chapter 9

Related work

9.1 Network and system co-design.

Combining system design with network elements is not a new idea [10, 17, 25, 26]. With the recent advancements in networking, such as software defined networking (SDN) and programmable switches, there has been a stronger call for network integrated systems.

Programmable switches allow for more flexibility in parsing packets, allowing for rapid prototyping of new protocols. There is also an added advantage of moving some compute to the switch itself [17, 26, 27]. Recently, NetCache provided fast key-value store caching layer in the switch at line rate [17]. NetCache physically stored the key-value cache in data- plane memory [17]. We do not take this approach as we only perform memory access control updates in the switch data-plane. All other computation is done in the control-plane.

One intuitive way of using SDN to solve failures is to re-route the traffic. Previous work extended SDN controllers to reroute traffic when links or switches fail [22, 25, 32]. Albatross, not only re-routes around partitions, but it enforces them by killing the partitioned node [25]. We enforce fate-sharing in a similar way to Albatross by dropping all packets going to and from the failure domain. We expand upon this principle to provide access control for shared memory in the switch data-plane.

9.2 Context-based RPC.

There has been a large amount of work customizing or improving RPC for particular environments [18, 39, 41, 42]. Similar to our work, Stuedi, et. al. integrated RPC with remote memory access (RDMA) for datacenter environments [41]. Kalia, et. al. improve upon that work by using two-sided datagram RDMA [18] and Su et. al. develop a new RDMA paradigm to achieve even better performance under RPC. Our work goes a step further by leveraging functionality in the network to achieve stronger guarantees instead of focusing on improving performance.

RPC has also been studied in the context of multi-processor operating systems [7, 11, 12, 35]. Paradigm [12], Hive [11], and Sprite [35] all utilize RPC as a mode of IPC between different kernels or processors. Hive, in particular, focusing on fault mitigation for the processor shared memory. This is orthogonal to our work, as they use RPC to communicate but do not consider RPC failure or recovery, they only focus on fault containment for memory [11]. More recently, Barrelfish, provides a multi-kernel abstraction for multi-processor systems, with the inter-kernel communication being handled by a user-level RPC library. Their implementation differs in two ways: 1) RPCS are not set over a network, but shared memory and 2) they do not consider partial failures [7]. In a disaggregated system, performing a network RPC is identical to writing out to shared memory, therefore we focus on a network based RPC implementation. We also consider RPC failure cases and recovery strategies.

Chapter 10

Conclusion

We have designed and prototyped a system porting RPCS to a disaggregated context with network managed memory protection and *exactly-once* semantics. Bifröst achieves promising performance results, performing slightly worse than Thrift without optimizations. Our results show that the majority of time spent in Bifröst is due to DSM accesses, whereas Thrift's latency is due to the reply from the server to the client. In our future work, we plan to address Bifröst's performance challenges by making several improvements to our system and evaluation.

We believe that, to reap the full benefits of disaggregation, we must take a holistic approach to designing systems on disaggregated racks. In particular, we must consider the role of the network as a critical piece in the design. This work is one of the initial steps in realizing the benefits of disaggregation by co-designing with the network.

Bibliography

[1] Intel, Facebook Collaborate on Future Data Center Rack Techologies, 2013. https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies/. → page 7

[2] rpclib - modern msgpack-rpc for C++, 2016. http://rpclib.net/. → page 9

[3] Thrift, 2017. https://thrift.apache.org/. → pages 3, 9, 22

[4] Barefoot Technology, 2018. https://www.barefootnetworks.com/technology/. → page 8

[5] gRPC, 2018. https://grpc.io/. → page 9

[6] K. Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computing. FAST '14. → page 7

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009. → pages 5, 41

[8] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 1984. → page 2

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 2014. → page 23

[10] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, 2017. → pages 1, 5, 7, 19, 38, 40

[11] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, 1995. → pages 5, 41

[12] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. Paradigm: a highly scalable shared-memory multicomputer architecture. *Computer*, 1991. → page 41

[13] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17. → page 8

[14] C. Constantinescu. Teraflops supercomputer: architecture and validation of the fault tolerance mechanisms. *IEEE Transactions on Computers*, 2000. → page 5

[15] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. Fault prediction under the microscope: A closer look into HPC systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012*, SC '12. → page 4

[16] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, 2016. → pages 1, 7

[17] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. → page 40

[18] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16. → page 41

[19] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, 1994. → page 9

[20] G. Kola, T. Kosar, and M. Livny. Faults in Large Distributed Systems and What We Can Do About Them. In *Euro-Par 2005 Parallel Processing*. → page 5

[21] S. Krishnapura, S. Achuthan, V. Lal, and T. Tang. Disaggregated Servers Drive Data Center Efficiency and Innovation. Technical report, Intel Corp., 2017. → page 1

[22] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić. Automatic Failure Recovery for Software-Defined Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, 2013. → page 40

[23] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets-IX, 2010. → page 26

[24] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15. → page 9

[25] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming Uncertainty in Distributed Systems with Help from the Network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015. → page 40

[26] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, . → page 40

[27] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, . → page 40

[28] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009. → page 7

[29] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, 2012. → page 7

[30] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, 1981. → page 9

[31] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, 2015. → page 9

[32] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. Vahdat, and R. N. Mysore. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, 2009. → page 40

[33] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 1991. → page 9

[34] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.* → page 9

[35] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *Computer*, 1988. → page 41

[36] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *International Conference on Dependable Systems and Networks, 2004*, DSN '04. → page 4

[37] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 1984. → page 6

[38] B. Schroeder and G. A. Gibson. A Large-scale Study of Failures in High-performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06. → page 4

[39] M. D. Schroeder and M. Burrows. Performance of the Firefly RPC. *ACM Trans. Comput. Syst.*, 1990. → page 41

[40] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. Feng Shui of supercomputer memory positional effects in DRAM and SRAM faults. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013*, SC '13. → page 4

[41] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle. DaRPC: Data Center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14. → page 41

[42] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. → page 41

[43] Y. Zhang, M. S. Squillante, A. Sivasubramaniam, and R. K. Sahoo. Performance Implications of Failures in Large-scale Cluster Scheduling. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP '04. → page 4