DISTRIBUTED PROCEDURE CALL

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Collin Shing-Tsi Lee
June 2021

This dissertation is online at: http://purl.stanford.edu/nv119rp9547

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Balaji Prabhakar

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Keith Winstein

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Communication performance is a critical factor when building distributed systems; communication costs can account for a non-trivial fraction of an application's service time. This has motivated many advances in raw communication performance, including both increased throughput and decreased latency. Unfortunately, these efforts ignore a more fundamental drain on communication performance: today's most common communication patterns, which result from the use of Remote Procedure Calls (RPCs), are inefficient and create otherwise unnecessary communication. With a more flexible communication pattern, it is possible to eliminate unnecessary communication, reduce the number of network hops, and ultimately improve the end-to-end latency of distributed executions.

To that end, this dissertation presents Distributed Procedure Call (DPC), a new communication primitive designed to support a "multi-hop" communication pattern for distributed systems. Multi-hop communication breaks the one-to-one mapping between requests and responses in traditional RPCs. With DPC, a server can delegate a request to one or more additional servers before responses are sent, forming a tree of requests. This allows DPC to reduce latency by eliminating messages and network hops. Compared to RPC, DPC can reduce end-to-end communication latency by 18% to 49% in a variety of applications without harming throughput.

DPC makes it easy for applications to take advantage of multi-hop communication. DPC's simple and general-purpose abstraction allows applications to dynamically form distributed executions. Behind this abstraction, the DPC protocol manages the multi-hop communication, matching request and response messages, and providing DPC completion and failure detection. Furthermore, DPC can be implemented as a library. This shields applications from the complexities of multi-hop communication and allows applications to focus on the logic of their distributed execution. DPC gives developers a new tool to build and optimize distributed systems.

Acknowledgments

I am grateful to my advisor, John Ousterhout. John taught me many things, both through his instruction and by his example. As an engineer, I am a product of his ideas on software design, and as a researcher, I am a product of his relentless pursuit of understanding. He instilled in me the importance of concepts and of clarity, ideas that apply as much to writing and presentations as it does to system design and programming. I'm thankful for the freedom he gave me to explore, but also for encouragement to stay focused to avoid straying too far afield. There was far more that he could teach than I could learn, but I'm thankful for every bit I got.

I want to thank my reading committee, Balaji Prabhakar and Keith Winstein, for encouraging me to focus on the aspects of my work that would become DPC. Thanks to Balaji for pointing out the connections between DPC and real systems in the wild; and thanks to Keith for helping me think about how my design works in different environments and suggesting interesting design alternatives like the "weighted acknowledgments" mechanism. I also want to thank, Mendel Rosenblum, who, in addition to serving on my oral defense committee, was a constant source of wisdom and he would drop by the office with interesting stories and anecdotes.

Special thanks to Jonathan Ellithorpe, Sadjad Fouladi, Yilong Li, and Diego Ongaro for the various discussions that contributed to the DPC work.

I'm also grateful for my labmates who were a huge part of my journey; they all supported me in many ways both big and small. Ryan Stutsman took me under his wing when I first joined the lab. Stephen Rumble gave me his desk (and what he claimed was not his plant). I had many "syncs" with Diego Ongaro. I could always count on Ankita Kejriwal and Jonathan Ellithorpe for their encouragement and positivity. Behnam Montazeri and Yilong Li were often hilarious even when they didn't intend to be; also, their work on Homa ultimately lead me to work on DPC. I appreciated the philosophical conversations I had

with Henry Qin. Despite my undirected grublings, I very much enjoyed my time working together with Seo Jin Park on RIFL. And, I'm glad I always found something random to think/talk/laugh about with Stephen Yang. Together, there were always interesting ideas to discuss and fun to be had. Thank you all for all the fond memories.

I also must thank my undergraduate professors Darren Atkinson, Shoba Krishnan, Sally Wood, and Katie Wilson whose encouragement and help lead me down the path of pursuing a Ph.D. Additionally, thanks to professors John Pauly and James Leckie who also helped me with my graduate school applications.

And finally, I am grateful to my parents, Joesph and Christinia, for instilling in me the curiosity to explore and the love of learning; their unwavering love and support have kept me grounded. Thanks to my sister, Nicolle, for always being there when I needed her and also for generally being the responsible sibling. And, thanks to Heymian for reading drafts of this dissertation, listening to my ramblings, and sharing in my successes and failures.

Thank you to everyone who helped make my journey possible; I could not have done it without you.

I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

—Abraham Maslow
The Psychology of Science

Contents

Abstract **iv**

Acknowledgments **v**

1 Introduction **1**

2 Motivation **6**

2.1 The Benefits of Multi-Hop 6

2.2 Multi-hop's Challenges 8

2.3 Distributed Procedure Call 9

3 The Abstraction **11**

3.1 DPC in Concept 11

3.2 Interface 13

3.3 Program Structure 14

3.4 Failure Semantics 15

3.5 Extensions 17

3.5.1 Execution Abort 17

3.5.2 Tracing and Debug 17

3.5.3 Exactly-Once Semantics and Transactions 17

4 The Protocol **19**

4.1 Identifiers 19

4.1.1 DPC Identifiers 19

4.1.2 Message Identifiers 20

4.2 Reply Address 21

4.3 Completion Detection . . . 21
4.3.1 Optimizing Manifests . . . 23
4.3.2 Implications . . . 25
4.4 Failure Detection . . . 26
4.4.1 Server Discovery . . . 27
4.4.2 Request Discovery . . . 28
4.4.3 Ping-Pong Mechanism . . . 29
4.4.4 Garbage Collection . . . 31
4.4.5 Handling Other False Positives and Negatives . . . 33
4.5 Execution Abort . . . 33
4.6 The Transport Below . . . 34
4.7 Summary . . . 35

5 The Details of Implementation **37**
5.1 Object-oriented API . . . 38
5.1.1 `Socket` . . . 38
5.1.2 `InMessage` . . . 40
5.1.3 `RooPC` . . . 43
5.1.4 `Request` . . . 43
5.2 Decomposition . . . 45
5.3 Request and Response Identifiers . . . 46
5.4 Data Structures . . . 47
5.5 Thread-safety . . . 49
5.6 Memory Management . . . 50
5.7 Summary . . . 51

6 Evaluation **52**
6.1 Benchmark Setup . . . 52
6.2 Basic Performance . . . 53
6.3 Distributed Ray Tracing . . . 55
6.4 Graph Queries . . . 58
6.5 Throughput and Resource Utilization . . . 61
6.6 Summary . . . 65

7 Discussion **66**
7.1 Practical Implications of Coupling 66
7.2 WAN and High-overhead Environments 67
7.3 Reliability and Retry in Unreliable Environments 68
7.4 Alternatives to Manifests 69
7.5 Expanding on DPC 71
7.5.1 Programming Patterns and Interface Definition Language 71
7.5.2 Scheduling 72
7.5.3 Distributed Pipe 73

8 Related Work **74**
8.1 Communication Primitives 74
8.2 Limits of the RPC Abstraction 76
8.3 Higher-level Frameworks 77
8.4 Raw Performance Improvements 78

9 Conclusion **79**
9.1 A Call to Develop New Tools 80
9.2 Toward General Distributed Computing 81
9.3 Final Comments 82

Bibliography **83**

List of Tables

6.1 Specification of machines used in benchmarks 53
6.2 Description of benchmarked graph queries 58
6.3 Message counts for benchmarked graph queries 61

List of Figures

1.1 Indexed lookup communication pattern 2

2.1 Multi-server communication patterns 7

3.1 DPC API 12
3.2 Pseudocode implementing Chord using the DPC API 14
3.3 Pseudocode comparing nested vs tail recursion 15

4.1 Example of a manifest's construction 23
4.2 Example of a manifest's fixed-size encoding 24

5.1 `Socket` abstract class definition 39
5.2 `InMessage` abstract class definition 41
5.3 `RooPC` abstraction class definition 42
5.4 `Request` abstract class definition 44

6.1 Latency vs hop count 54
6.2 Latency vs degree of fan-out 55
6.3 Distributed ray tracing communication pattern 56
6.4 Latency of distributed ray tracing pattern 57
6.5 Latency of graph query patterns 60
6.6 CPU usage for graph query patterns 62
6.7 Latency vs throughput of two graph query patterns 64

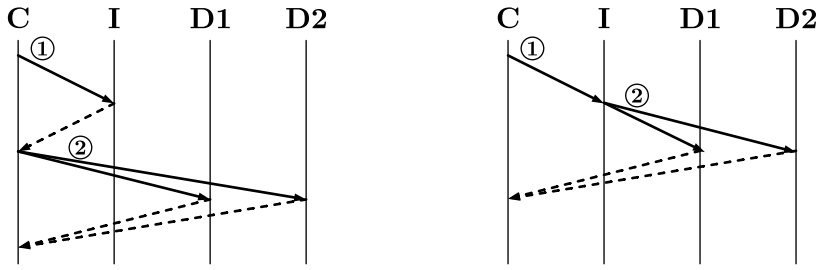
Chapter 1

Introduction

Communication is a fundamental building block of distributed systems. Building a distributed system involves both deciding how data and logic are split across multiple machines and also defining how the machines will coordinate and communicate. Unfortunately, building distributed systems using low-level communication primitives can be inherently complex and tedious. In response, higher-level abstractions, frameworks, and tooling have been developed to hide complexity and streamline development. Today, it is hard to imagine building a complex distributed system without the help of higher-level communication primitives that shield programmers from the nuts and bolts of communication and allow developers to express communication in a form that matches closely with the application logic.

The de facto standard communication primitive for building distributed systems is Remote Procedure Call (RPC) [11]. RPC encapsulates a style of communication between two machines. It is modeled as a request and response message pair between a client and a server and involves one network round trip per invocation. With RPC, a client sends a request to a server where the server processes the request and returns a response back to the client. RPC's simple abstraction fits well with the procedural way of thinking that many developers have; it presents itself as function invocation, except that the function actually executes on a remote machine. Additionally, there are many robust and general-purpose implementations and frameworks for the RPC model [6, 2, 3]. This allows developers to adopt the RPC model without needing to build their own implementation.

RPC fits a model where a client operation can be completed by a single server. Unfortunately, as distributed systems have become larger and more complex, there are more



(a) RPC **(b)** Multi-hop (DPC)

Figure 1.1: The communication pattern for an indexed lookup of multiple documents in a distributed document store using (a) RPC and (b) multi-hop (DPC). In step (1), the index server (I) is queried for the document ids. In step (2), multiple documents are fetched from the document servers (D1, D2).

and more situations where a single logical operation might involve multiple servers. Furthermore, some servers may be dependent on information provided by other servers. Take for example a searchable document store. This system would not only hold the documents themselves but also maintain an index for supporting search. To scale, the system would likely distribute the document and index data across multiple machines. As a result, an indexed-lookup operation in this system must first ask at least one server to consult the index information and then use that information to retrieve the documents from potentially a number of other servers.

RPC can be used to implement these complex operations with dependencies across multiple servers, but using RPC in this way can be inefficient and ultimately result in higher end-to-end latency for the operation. Figure 1.1(a) shows the communication pattern of the indexed document lookup example as implemented using RPCs. Here two serialized sets of RPCs are used: the first RPC is used to perform an index lookup, and the second set of one or more RPCs is used to retrieve the identified documents. In this example, the result of the index lookup is only used to form the subsequent document retrieval requests. It is logically unnecessary to send an intermediate response back to the client, just to be reformed into new requests to be sent out again. However, the RPC model imposes this communication pattern and thus creates inefficiency.

Without RPC, services might naturally form a different communication pattern where control hops from one server to the next without constantly returning to some central node. This can be seen in Figure 1.1(b). In this world, intermediate results would be processed

locally on the server that generated the result to form a new follow-up request that the server itself sends directly. This *multi-hop* communication pattern would eliminate the sending of intermediate results and reduce end-to-end latency.

Although many applications could benefit from adopting a multi-hop style of communication, there is currently no simple way for applications to do so. To use multi-hop communication today, an application must build and maintain its own multi-hop protocol, likely on top of some form of message passing. Such ad hoc approaches force applications to handle a wide range of issues such as matching incoming response messages to their request and detecting whether a multi-hop execution has completed or failed. While some uses like Chain Replication [35] may deem the multi-hop pattern important enough to merit the cost of an ad hoc implementation, for most applications it is simply easier to forgo the benefits of adopting a multi-hop pattern and use some well maintained RPC-style library.

This gap between the potential benefits of leveraging multi-hop communication and the difficulty of implementing a multi-hop execution in practice led me to ask a series of questions. How could I make it easier for applications to adopt multi-hop communication? Can the difficult aspects of using multi-hop be implemented as a reusable library? Is there a general way to solve the problems of detecting whether an execution has completed or failed? How can this be done when the execution is distributed with no single point of coordination? Are the general solutions efficient enough to not erode the natural benefits of multi-hop communication? And finally, can all of this complexity be hidden from applications? This dissertation is my answer to these questions.

This dissertation describes *Distributed Procedure Call* (DPC), a new communication primitive that encapsulates the implementation of multi-hop communication within a simple abstraction. The DPC abstraction models the distributed execution supported by a multi-hop communication pattern. The abstraction logically ties together the messages used in an execution and solves the problem of matching request and response messages. DPC also defines a protocol that detects a DPC's completion or failure. The protocol runs below the abstraction and is completely hidden from applications. Finally, DPC can be implemented as a library, giving applications an easy way to take advantage of the benefits of a multi-hop communication pattern.

There are three notable features of Distributed Procedure Call:

- **Simple and General-purpose Multi-hop Abstraction**: DPC is designed to support arbitrary multi-hop communication patterns with just three client-side and four

server-side API calls. Its simple API can be used to express communication patterns with any number of consecutive server visits ("hops") and any number of parallel server visits ("fan-out").

- **Improved Latency**: By making multi-hop communication accessible, DPC allows applications to decrease their end-to-end latency. DPC eliminates messages from the critical path, reducing both the number of network hops that an application execution must wait for as well as the number of messages that must be processed.

- **Lightweight Protocol**: The DPC protocol is lightweight enough to be implemented efficiently. This allows high performance DPC implementations to be used in low latency environments with single digit microsecond network round-trip times.

To demonstrate and evaluate DPC, I built Roo, a reference DPC implementation. Roo not only provided an implementation of DPC that could be benchmarked but also served as a platform to test and verify DPC protocol designs. I found that Roo, and thus the DPC abstraction, was able to support a wide variety of application execution patterns. Even in already low latency environments, Roo reduced the latency of complex execution patterns by anywhere from 18% to 49% versus a comparable RPC implementation. Additionally, Roo provides these latency reductions without harming throughput.

This work makes the following main contributions:

- It defines Distributed Procedure Call, a new communication primitive to support multi-hop communication.

- It defines a protocol that solves the two major challenges in multi-hop execution: detecting execution completion and detecting execution failures.

- It demonstrates that DPC can be implemented with a clean and simple API that supports a variety of communication patterns efficiently.

- It shows that using DPC can significantly reduce end-to-end latency for a variety of applications with Roo showing 18% to 49% reductions in latency in already low-latency environments.

- It provides Roo, a complete and high-performance reference implementation of DPC and demonstrates that the DPC protocol can be implemented efficiently.

The remainder of this dissertation explains the motivation and potential use cases for DPC (Chapter 2); covers the DPC abstraction from an applications perspective (Chapter 3); describes the underlying DPC protocol (Chapter 4); provides a variety of Roo's implementation details that may be of interest to others building a DPC implementation (Chapter 5); evaluates both the benefits of multi-hop and the performance of Roo (Chapter 6); and discusses additional DPC topics and related work (Chapters 7–8).

Chapter 2

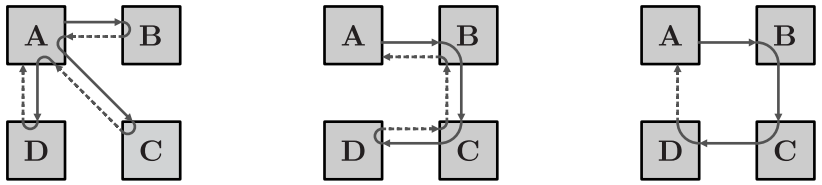
Motivation

DPC is an abstraction for multi-hop communication and is designed to address the main challenges of using multi-hop. This allows developers to more easily build applications using the multi-hop pattern. Section 2.1 discusses why and which applications might want to use multi-hop communication and Section 2.2 describes why it might be difficult for applications to create multi-hop executions without the help of DPC.

2.1 The Benefits of Multi-Hop

Many applications today are demanding lower and lower end-to-end latency. For distributed applications, latency is often limited by the cost of communication. As a result, an increasing amount of engineering effort has focused on optimizing and accelerating the network stack [4, 19, 30, 9, 17, 28, 10, 20]. However, a distributed application's communication latency is ultimately a function of the number of networks hops needed to complete its execution; the more times an operation needs to cross the network the longer its execution will take.

Today, most distributed services structure their communication using the RPC pattern. When the execution of a service request involves multiple machines in sequence, there are two common RPC-based communication patterns. The first and most common is the hub-and-spoke pattern of RPC usage where a single server A makes separate RPCs in series to multiple servers (B, C, D) involved in request processing. This results in the pattern of communication shown in Figure 2.1(a). The second is the nested RPC pattern where server A makes an RPC to server B and server B invokes a nested RPC to server C and so on. This results in the pattern shown in Figure 2.1(b). In both cases, the minimum latency is



(a) Hub-and-spoke **(b)** Nested **(c)** Multi-hop

Figure 2.1: Communication patterns involving multiple sequential server visits.

n times the network round-trip time where n is the number of servers involved.

However, RPC-based communication patterns can be suboptimal for many applications. This is true for executions where the intermediate results are only used to form a subsequent request, for example in Figure 2.1(a) where the responses from servers B and C are only used to form the requests to servers C and D respectively. In these cases, a multi-hop pattern would be better suited. The multi-hop communication pattern, shown in Figure 2.1(c), gives applications optimal communication latency by eliminating unnecessary messages and network hops from the critical path.

There are many examples of services that could naturally leverage a multi-hop communication pattern. Here are six:

1. **Proxied Service** — Consider a service with a proxy or middlebox between the client and the server; the middlebox gives the service control over how client requests are routed. When a client sends a request to the middlebox, the middlebox decides which server should handle the request and forwards the request to that server for processing. This can be used in a number of ways including load balancing and allowing the service to seamlessly swap in and out servers for migration and failure recovery.

2. **Indexed Lookup** — Consider an object store with secondary indexing [21]. An indexed lookup would first make a request to the server containing the indexing information and then forward the request to the servers containing the objects found.

3. **Chord Distributed Lookup** — The Chord [34] distributed lookup protocol is a building block for peer-to-peer distributed hash tables which maps a given key to the node that owns the key. There is no centralized mapping information in the Chord protocol. Instead, the lookup involves searching a number of nodes in sequence to find the owner of a given key.

4. **Chain Replication Updates** — Chain Replication [35] defines a protocol for updating the backup servers of a fault-tolerant storage system. Storage updates in this protocol flow from the client through the primary and backup servers in a sequential chain before the last server in the chain returns directly back to the client.

5. **Distributed Ray Tracing** — Distributed ray tracing [37, 16, 7] is a technique for rendering a 3D image using a cluster of machines by simulating light as it bounces through objects in the scene. Computing a ray bounce requires sequentially traversing scene data distributed across many nodes with the process repeating for each bounce of the light ray. The number of hops is proportional to the number of bounces that need to be computed.

6. **Interactive Graph Query** — Consider a graph database used to store social network data [8]. A graph query might involve returning a recent post, all replies to that post, and metadata associated with the replies like the authors' name. Each traversal in the graph generates a set of requests; first to get the recent post, then to get the set of replies, and finally to collect the metadata for the replies. The number of sequential sets of requests is proportional to the depth of the graph traversal.

2.2 Multi-hop's Challenges

While many applications could benefit from using multi-hop communication patterns, multi-hop is not widely used. One likely reason is that applications wishing to adopt a multi-hop style of communication would need to implement and support their own ad hoc protocol. With the engineering effort this might entail, adopting multi-hop could be a non-starter for many use cases.

There are three main challenges facing applications trying to directly implement multi-hop communication. The first and simplest challenge involves allowing a client to match request and response messages in a multi-hop execution. Without this ability, a client execution could mismatch responses and return incorrect results when multiple requests are issued concurrently. This challenge is similar to matching an RPC response message to its RPC request message. However, with multi-hop communication, each request may generate multiple responses. Furthermore, responses could arrive from servers other than the target of the initial request. For example, a client that initiates an indexed object

lookup by sending a request to an index server could receive multiple response messages from multiple object servers. In addition to simply identifying an incoming response's associated request, it might also be necessary to dispatch and deliver the response to the thread or context responsible. While solving this challenge is relatively simple, it still requires extra mechanisms to be built and maintained.

A more significant challenge for multi-hop communication is determining the set of response messages a client should expect for a multi-hop execution. Without this information, a client cannot know when all responses have been received and thus cannot know whether or not an execution is complete. A client does not necessarily know how many response messages to expect because the multi-hop topology can be dynamically generated by the servers processing the distributed execution. Servers can send messages directly from server to server and can send messages to multiple servers in parallel (i.e. fan-out). Any of these servers can also send response messages back to the client. As a result, no single node, including the client, knows either the full communication topology or even the set of sent responses. Supporting dynamic multi-hop executions will require implementing a mechanism to discover the set of expected response messages and detect execution completion.

Similarly, it is also difficult for a client to know whether or not a multi-hop execution has failed. Typically, a client might check with a server if a request has not been completed in some time. However, because of the nature of the multi-hop communication pattern, clients may not know which servers are executing related requests and thus would not know which servers to check. To detect failures, a mechanism is needed both to discover the servers involved and to check them for failures.

Any implemented solutions for supporting multi-hop communication would also need to be efficient. The primary reason to adopt the multi-hop pattern is to improve end-to-end latency. If, however, the overhead of supporting response matching, completion detections, and failure detection exceeds the communication pattern's natural latency reductions, there is no reason to adopt multi-hop communication. As a result, directly adopting multi-hop introduces not only logical challenges but also practical performance challenges.

2.3 Distributed Procedure Call

Given the relative difficulty of directly adopting multi-hop communication patterns, it is no surprise that the use of multi-hop is relatively rare. While there may be cases where

only a subset of multi-hop's challenges need to be solved, having to implement and support solutions for any one of them may already be deemed "too hard" or "not worth the effort." Without a turn-key solution, it seems unlikely that multi-hop communication patterns will be widely adopted.

The goal of DPC is to provide an easy way to use the multi-hop communication pattern. DPC exposes a simple abstraction that hides the complexities and challenges of a multi-hop protocol. DPC can thus be implemented as a library, centralizing and amortizing the effort of implementing and optimizing the multi-hop protocol. Applications can then leverage a DPC library implementation to gain the performance benefits of a multi-hop communication pattern without significant effort. Ultimately, I hope that DPC libraries can both simplify existing uses of multi-hop communication and also allow more applications to take advantage of the communication pattern's benefits.

Chapter 3

The Abstraction

Distributed Procedure Call is an abstraction that encapsulates the multi-hop communication pattern. This chapter covers the abstraction from the perspective of an application using DPC. This includes a general API, possible failure semantics, and a variety of other relevant concepts.

3.1 DPC in Concept

Distributed Procedure Call is designed to provide a simple abstraction for executing any multi-hop communication pattern. The DPC abstraction shares many similarities with RPC. This gives developers using DPC a shared vocabulary and a sense of familiarity that may help with understanding. Both RPC and DPC represent a single high-level operation with remote execution. Like RPC, DPC's core primitives are request and response messages. Request messages are sent to servers and response messages are sent back to the client. Also, like RPC, DPC solves the problem of associating request messages with response messages. However, where a single RPC is processed by a single server and has a single request and a single response, a single DPC can be processed by multiple servers, with multiple requests and multiple responses

There are two ways in which a DPC can distribute processing across multiple servers. First, a DPC can be distributed to multiple servers in series. As with RPCs, a DPC client can send a request to the server, which can process the request and return a response back to the client. However, unlike the RPC server abstraction, a DPC server can choose to delegate the responsibility of completing the request to a different server by issuing

CLIENT API

send(*handle*, *destination*, *request*) → *handle*

Send a *request* message to the *destination* as part of the DPC and return a *handle* to the associated DPC. If a *handle* is provided, the request is added to the associated DPC. Otherwise, a new DPC is started.

receive(*handle*) → *response*, *status*

Get a received *response* message for the DPC referenced by the *handle*, if a message is available, as well as the current status of the DPC (see **status()**).

status(*handle*) → *status*

Get the current *status* for the DPC referenced by the *handle*. The *status* can be either `IN_PROGRESS`, `FAILED`, or `COMPLETE`.

SERVER API

receive() → *request*, *handle*

Get a received *request* message and associated *handle* if a message is available.

reply(*handle*, *response*)

Send a new *response* message back to the client as part of processing the request associated with the *handle*.

delegate(*handle*, *destination*, *request*)

Send a new delegated *request* to the *destination* as part of processing the request associated with the *handle*.

finish(*handle*)

Declare the processing of the request associated with the *handle* complete and perform any necessary cleanup.

Figure 3.1: Description of the DPC client and server API.

a *delegated request*. Furthermore, servers processing delegated requests can themselves delegate additional requests. In this way, delegation can reach any arbitrary depth and support a communication pattern with multiple "hops."

The second way a DPC can be distributed is to multiple servers in parallel. This is supported by allowing a DPC client to send any number of requests as part of a DPC and allowing servers to send any number of delegated requests to other servers. This allows DPC to form a tree of requests with any degree of "fan-out" from any server node in the multi-hop execution. Additionally, each node in the execution can choose to send any number of responses back to the client. As a result, the client models a DPC as a collection of one or more outgoing requests and any number of incoming responses.

3.2 Interface

The DPC client API, shown in Figure 3.1 is similar to that of an asynchronous RPC abstraction, except that there may be multiple requests and responses associated with a single DPC. To initiate a new DPC with its first request, the client calls `send()` without specifying a DPC *handle*; the call returns a new *handle* to be used in subsequent calls. To send another request as part of the same DPC, `send()` can be called again with the *handle* specified. To receive a response for a DPC, the client calls `receive()` specifying the *handle*. To ensure the client receives all responses, the client should repeatedly call `receive()` until a *status* of `COMPLETE` is returned.

Figure 3.1 also shows the DPC server interface. To process requests, the server first calls `receive()` which returns a request message and associated *handle*. While processing the request, the server can either send a response by calling `reply()` or send a delegated request by calling `delegate()`. Both `delegate()` and `reply()` can be called multiple times to send multiple delegated requests and responses respectively. Finally, `finish()` is called to indicate that request processing is complete.

DPC's API is asynchronous by design for two reasons. First, the asynchronous API gives applications more flexibility. With an asynchronous interface, clients can send requests as they become ready and can dynamically decide what requests to send. In contrast, a synchronous interface might require all requests to be sent at once and block the client until all the expected responses have been received; this also results in a more complex API as the client must be able to supply multiple requests and receive multiple responses in a single call. Second, the asynchronous DPC interface allows for better pipeline parallelism and thus potentially better performance. For example, a client can process one received response while waiting for other responses to arrive. This allows the client to do useful work while waiting for all the responses to arrive. This can reduce the end-to-end latency of a DPC. Of course, if an application prefers a synchronous API, the asynchronous API could be trivially wrapped to provide this behavior.

Figure 3.2 shows how the DPC API can be used to implement the Chord [34] lookup protocol. The use of the API fits naturally into the code flow. By hiding the complex multi-hop protocol implementation behind the DPC API, the application code remains simple and focused on the application logic.

```
function FindOwner(key)
    // Start by asking any node for the owner of the key
    node ← SelectNode()
    dpcHandle ← Client::send(None, node, {key})
    // Wait for a response
    {owner}, status ← Client::receive(dpcHandle)
    // status can be checked to implement error handling
    return owner
```

(a) Client Code

```
function ServiceRequest()
    // Get an incoming FindOwner request
    {key}, requestHandle ← Server::receive()
    if KnowsOwner(key) then
        owner ← OwnerOf(key)
        Server::reply(requestHandle, {owner})
    else
        // Ask a "closer" node to service the request
        node ← ClosestKnownNode(key)
        Server::delegate(requestHandle, node, {key})
    Server::finish(requestHandle)
```

(b) Server Code

Figure 3.2: Pseudocode implementing the Chord [34] distributed lookup protocol using the DPC API. The protocol maps a given key to the node that owns the key. Each node in Chord holds partial mapping information. The key lookup algorithm searches a number of nodes serially. Each node directs the search to a node that is "closer" to the owner until the owner is found. For clarity, the pseudocode omits and simplifies the details of the protocol that are unnecessary to show how the algorithm would use the DPC API.

3.3 Program Structure

To take advantage of DPC's benefits, application code must be structured in a manner similar in concept to a series of *tail calls*. With a tail call, the calling context never touches the return value of the call so the value can be returned directly to the caller's parent context. When a program consists of a chained series of tail calls, as with tail recursion, the return value of the deepest call in the chain can be returned directly to the initial calling context. A DPC delegated request is similar to a tail call. Servers sending a delegated request will not receive a response. Instead, responses will go directly back to the client. Where DPC differs is that it allows applications to make multiple requests in parallel and allows any server in the chain to send both delegated requests and responses.

```
function SumList(node)
    if node is last list element then
        reply node.val
    else
        val ← invoke SumList(node.next)
        sum ← val + node.val
        reply sum
```

(a) Nested Call

```
function SumList(node, sum)
    if node is last list element then
        total ← sum + node.val
        reply total
    else
        sum ← sum + node.val
        delegate SumList(node.next, sum)
```

(b) Tail Delegation

Figure 3.3: Pseudocode showing a distributed algorithm using both nested recursion and tail recursion. The algorithm sums the values contained in distributed linked list of *node* elements. Each *node* contains a value (*node.val*) and identifies the next element in the list (*node.next*). The keywords **reply**, **invoke**, and **delegate** indicate where communication occurs. In Figure 3.3(a), RPC-style communication is used; **invoke** sends a request and waits for a response while **reply** sends a response to the node that sent the request. In Figure 3.3(b), DPC-style communication is used; **delegate** simply sends a new request and **reply** sends a response back to the initial client.

DPC can be used by any application that can be structured in a tail-call style of execution. For applications that are naturally structured in this way, adopting DPC is easy. The Chord lookup protocol (shown in Figure 3.2) is one such example as it is naturally tail-recursive. For other applications, using DPC requires additional techniques similar to those used to convert algorithms without tail calls into ones with them. This might involve passing some additional state along with the call and shifting some of the logic between the callee's and caller's execution. For example, consider the logic for summing the values contained in a distributed, linked list. An implementation using nested RPCs, shown in Figure 3.3(a), might compute the sum as responses flow back up the call chain. To use a structure that benefits from DPC, shown in Figure 3.3(b), a running sum can be computed and passed along with a delegated request. This way the last hop can complete the computation and reply directly back to the client. Simple considerations like these allow an even wider range of applications to use DPC.

3.4 Failure Semantics

Failures create uncertainty. If a server crashes while servicing a request, the client cannot generally tell whether the request was received and acted on. While this uncertainty is impossible to eliminate, communication layers try to bound and precisely articulate the

range of possible behavior. This allows applications to react accordingly.

The most common failure model for communication layers is *at-most-once* semantics. At-most-once semantics guarantees that the execution of a request will never be duplicated. The semantics do, however, allow for the possibility that a request did not complete in the event of a failure. For example, consider at-most-once semantics in the context of RPC, which contains a single request. If a client detects that an RPC has failed, the client doesn't know if the request was executed or not. The client can be certain though that the request is never executed twice.

DPC adopts at-most-once semantics and guarantees that request execution is never duplicated. However, since DPC consists of multiple requests, DPC has a range of potential behavior in the event of a failure. When a client detects a DPC failure, it is possible for the DPC as a whole to have been completely executed, partially executed, or not executed at all. When thinking of DPC as a collection of requests, partial executions can occur when some but not all the requests are processed. Still, applications can be certain that no request within a DPC is ever executed more than once.

There are a few common ways to handle DPC failures. First, for some applications, partial execution is perfectly acceptable. For example, it would be fine for an inexact search query to have missing results. For these applications, the failure can simply be ignored (or more realistically logged). Second, when using DPCs that don't mutate state (e.g. a read-only query), applications can safely retry the entire DPC since partial executions don't cause any inconsistencies. Finally, if the DPC does mutate state, the application can retry the entire DPC but must filter out duplicate request executions by keeping around some information about previous executions [27].

Intuitively, one might think that the complexity of handling DPC failures should be greater than that of RPC because of DPC's broader range of possible behavior. However, the mechanisms needed to handle DPC failures, and thus the burden on developers, are the same for both RPC and DPC. Applications tolerant of partial DPC executions require no mechanism for failure handling. For "read-only" executions, both RPC and DPC require clients to retry the execution. Finally, for reties of stateful executions, both RPC and DPC require that applications be prepared to handle individual duplicate requests. Though in theory, DPC clients have less certainty about the execution state in the event of a failure compared to RPC, in practice, this does not make DPC failures more complex to handle since both DPC and RPC failures are ultimately handled in the same way.

3.5 Extensions

The previously described core aspects of DPC are the focus of this dissertation. However, there are ways that DPC can be extended to offer additional functionality. This section describes a few possible extensions.

3.5.1 Execution Abort

In some cases, an application may wish to abort a distributed execution once it has started. In general, a client initiating an abort must be able to contact the servers involved in the execution. Since DPC manages the communication between all components of a distributed execution, DPC implementations are well positioned to track the servers involved in the execution and deliver an abort signal to each server when requested. DPC implementations could hide the complexities of an abort protocol behind a single client API call. This is described in more detail in section 4.5.

3.5.2 Tracing and Debug

Debugging and tracing a distributed execution is a challenging problem. One of the challenges is collecting and associating the debug information from the multiple components of a distributed execution. Given that a DPC inherently associates the separate parts of a distributed execution, DPC would be a natural layer to provide instrumentation and tracing or provide the necessary context for better debug output. For example, DPC could provide identifiers and sequence numbers for logging that would allow the log output from multiple servers to be filtered and serialized, providing a unified and time-ordered trace of a DPC's execution. While this extension of DPC is not explored in this dissertation, I believe the possibility of better distributed debugging and tracing could be another benefit of DPC.

3.5.3 Exactly-Once Semantics and Transactions

In an ideal world, many applications may prefer to use DPC with stronger failure semantics like *exactly-once* semantics or even transaction semantics. Unfortunately, communication layers, in general, cannot guarantee these stronger semantics without relying on additional failure recovery mechanisms typically provided by the application layer. For example, the application might need to persist additional state about an execution along with any resulting application-specific mutations. For this reason, DPC is designed to provide at-most-once

semantics (as described in Section 3.4). This allowed DPC to remain as general as possible, requiring no special application support for fault tolerance. However, for applications that are willing to provide the extra support, it is possible to extend the basic DPC design to provided stronger failure semantics.

With exactly-once semantics, each request appears to have executed exactly once. Systems like NFS [12] and RIFL [27] support exactly-once RPC semantics by allowing RPCs to be retried, filtering out duplicate request executions, and responding to the duplicate requests with the results of the original execution. This requires systems to keep track of previously executed requests and their responses. To support exactly-once semantics for DPC, delegated requests must also be stored on request execution and resent in reaction to a duplicate request. This ensures that retries of a DPC will cause all request and response messages to be resent but will prevent any previously executed request from being reprocessed. To achieve exactly-once semantics, a failed DPC must simply be retried until it succeeds.

To completely support exactly-once semantics, the request and response metadata used to filter execution and resend messages must survive a server crash. More precisely, RIFL states that the metadata must be made durable atomically with any durable side-effects of the request execution. This ensures consistency in the event of a crash; if a server crashes while processing a request, either the request was executed and remembered, or the request side-effects are lost along with any memory of the request. Because DPC is only a communication primitive, it must rely on external support for durability. Additionally, exactly-once mechanisms also require support from the application to ensure that the metadata is atomically made durable with the application's side-effects. With application support, DPC can provide exactly-once semantics in the face of server crashes.

Using the exactly-once mechanisms as a building block, a DPC execution can also be extended to provide transaction semantics. RIFL details a distributed transaction protocol that leverages RIFL's exactly-once RPC mechanism. In RIFL's protocol, transactions involve a collection of RPCs supported by the exactly-once mechanism. A similar transaction protocol could be used with DPC where a single DPC with support for exactly-once semantics is used instead of the collection of RPCs in the RIFL transaction protocol. While the details of such a protocol are beyond the scope of this work, it is certainly an interesting area of future work.

Chapter 4

The Protocol

Behind the simple DPC abstraction, the DPC protocol shoulders the challenges and complexity of multi-hop communication. The protocol's responsibilities range from the straightforward, like ensuring a DPC's responses are correctly delivered, to the complex, like determining whether a DPC execution has completed or failed. The primary challenge for the DPC protocol is that a client does not know which servers will be involved in a DPC at the time of initiation; the DPC topology is formed dynamically by intermediate servers.

This chapter describes various parts of the robust and efficient protocol used to support the DPC abstraction. While parts of the protocol are complex, the intention is for the protocol to be implemented in a reusable DPC layer or library. This shields applications from all the complexity described in this chapter, which applications would otherwise need to handle themselves using ad hoc solutions.

4.1 Identifiers

The DPC protocol uses two types of identifiers: the *DPC id* and the *message id*. For both correctness and performance, these identifiers must be carefully generated and assigned. This section describes the protocol's requirements for id generation and how the ids are defined to meet these requirements.

4.1.1 DPC Identifiers

Each DPC is assigned a unique identifier. DPC ids are used to logically associate messages belonging to a single DPC and separate messages associated with different DPCs. This

includes both request and response messages as well as the protocol-level messages (introduced later in this chapter). Every message is tagged with its DPC's id. This allows the DPC layer to perform request and response message matching for applications and also allows all parts of the DPC protocol to operate on each DPC in logical isolation.

DPC ids must be "globally" unique; the scope of uniqueness must encompass all clients and servers that might communicate with each other for as long as DPC messages might be received by any of those clients and servers. That means ids need not be unique between two completely independent groups of machines and ids can be reused after a full cluster outage that guarantees no message from before the outage will be received after the outage. In practice, however, since full cluster outages are rare and it is often difficult to guarantee machines won't communicate, it is likely simpler to ensure uniqueness across the entire cluster for all time.

DPC ids are assigned by the client that initiated the DPC. Generating these ids can be done in a number of different ways, however, the protocol assumes that each client can independently generate globally unique identifiers. One way to accomplish this is to use a globally unique id prefix followed by a locally unique suffix (e.g. a sequence number). This in essence shifts the problem from generating globally unique DPC ids to globally unique client ids. However, the frequency and number of client ids generated is expected to be far smaller than that of DPC ids and thus it is likely acceptable to use a more expensive centralized method of client id assignment or use a large random number generator for probabilistically unique client ids. Notably, seemingly obvious candidates for the client id, like its machine's IP address, would not be appropriate; an IP address could be reused after a restart and result in non-unique DPC ids since previously used sequence numbers may also be reused.

4.1.2 Message Identifiers

The DPC protocol also uses per message identifiers for request and response messages; the use of these message ids will be explained later in this chapter. While message ids only need to be unique within a DPC for correctness, a number of performance optimizations, explained in Section 4.3, can be enabled if message ids are generated with additional constraints. In particular, the goal is to provide a unique and contiguous range of ids for the execution of each request. Request processing can then assign message ids from the provided range to all messages sent as part of the request's execution.

To create these unique id ranges, a message id is defined as the concatenation of three fields: a globally unique *node id*, a node-unique *execution context number*, and a context-unique *message number*. First, a globally unique node id can be assigned to each client or server in the same way client ids are assigned for DPC ids (in fact, the client id can be the node id). This provides each client or server with a range of ids that can be independently assigned. Next, each server will assign a new execution context number for each incoming request. On clients, a new context number is assigned for each new DPC. The execution context number further divides the id space and provides each request execution with its own unique range of message ids. An execution can then assign a unique message id from its range by assigning a different message number to each of its outbound messages.

For correctness, it is sufficient for executions to assign message numbers from the given range as it sees fit so long as assignments are unique. However, particular assignment schemes allow for certain performance optimizations. One aspect of this scheme involves using positives message numbers when assigning request message ids and negative message numbers when assigning response message ids. The full scheme and resulting performance optimization is described later in Section 4.3.

4.2 Reply Address

With DPC, each server must be able to directly send a response back to the client. Servers processing a DPC's request must therefore know the address of the client that initiated the DPC. While most point-to-point communication primitives like RPC can rely on the underlying transport like TCP to know where to deliver responses, DPC servers may not have this same implicit knowledge since requests may not always come directly from the client. Instead, every DPC request message explicitly carries the client's address. This way the DPC protocol running on the server will know where to send a response when the application wishes to reply.

4.3 Completion Detection

One of DPC's key features is that it allows clients to know when a DPC's execution has completed even though the execution is not driven by the client. Logically, a client knows that a DPC is complete once it has received all of the DPC's response messages. The

challenge for the protocol is determining the exact set of response messages that should be expected. This is because a DPC's execution topology is dynamically generated as servers process requests; the client is only aware of the requests it sent directly.

The DPC completion detection mechanism incrementally collects information about the DPC's execution topology and passes this information back to the client. The client can use this information to gradually discover the DPC's topology and use that to determine the exact set of response messages to expect. The completion detection mechanism is explained by answering three questions. First, what information is needed to allow the client to reconstruct the execution topology? Second, how does the client know if the reconstructed topology is complete? And lastly, how does the client use the topology information to safely detect DPC completion?

The topology information collected exactly mirrors each component that makes up the DPC execution. A DPC execution forms a tree of requests starting at the client. When a server processes an incoming request it can generate additional delegated requests as well as responses, dynamically growing the tree. When a server finishes processing an incoming request, it collects the following information, which will eventually be returned to the client: the processed incoming request's message id, the message ids of any generated delegated requests, and the message ids of any generated responses. This bundle of information is called the request's *manifest*. As an example, the manifest for a particular request execution is shown in Figure 4.1. Each request generates a corresponding manifest. A request's manifest also signals that the request has finished processing since the manifest is only generated after processing is complete and thus the information in the manifest is final.

A client collects a DPC's topology information by tracking the DPC's set of expected manifests and responses. Initially, the client only expects a manifest for each of the DPC's initial requests (i.e. those sent directly by the client), and the set of expected responses is empty. When the client receives a manifest, it may learn of previously unknown requests and responses. Using this new information, the client expands the set of expected manifests and responses accordingly. Once a client has received all expected manifests (i.e. a manifest for each request), the client becomes aware of a DPC's entire execution, including all expected responses. The client can never erroneously believe it has received every manifest because a newly received manifest will always indicate additional missing manifests (i.e. previously unknown delegated requests) if any exist.

When a client has collected a DPC's complete topology information, it will know the full

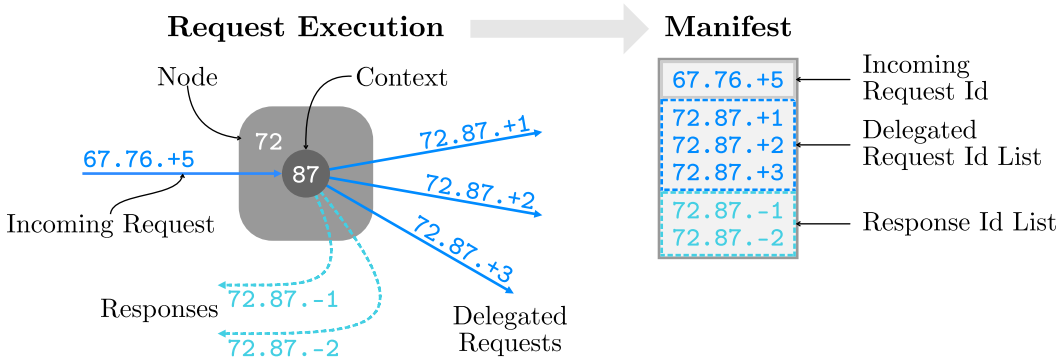


Figure 4.1: An example showing the processing of a single request within a larger DPC execution and the resulting manifest. The incoming request (`67.76.+5`) is received and processed by a server node (`72`), which assigns the execution a new context number (`87`). The execution sends three delegated requests and two responses; message ids are assigned based on the node id and execution context number as described in Section 4.1.2. The resulting manifest identifies the request that was processed as well as the delegated requests and responses that were sent as part of the execution. Omitted from this diagram for simplicity is the DPC id, which would be included in all messages.

list of response messages it should receive for the DPC. This means that once all manifests and also all response messages are received, the client can be sure that the DPC is complete.

4.3.1 Optimizing Manifests

In the initial design of the DPC protocol, manifests included variable-length lists of request and response message ids and servers sent manifests back to the client in dedicated messages after `finish()` was invoked. Unfortunately, experiments showed that both the variable-length manifests and the overhead of sending separate messages had a non-trivial impact on performance, particularly for larger DPCs. I subsequently modified the protocol to eliminate the separate manifest messages: manifest information is piggybacked on existing messages. I also introduced techniques to ensure the manifest information only adds a small and fixed number of bytes to existing messages, regardless of the request topology.

The first step is to make each manifest fixed-size, regardless of fan-out. To do this, each manifest must encode their variable-sized lists of request and response message ids in a fixed number of bytes. The idea is to carefully assign ids so that each of the request and response id lists can be encoded using a count. Recall that a message id consists of the

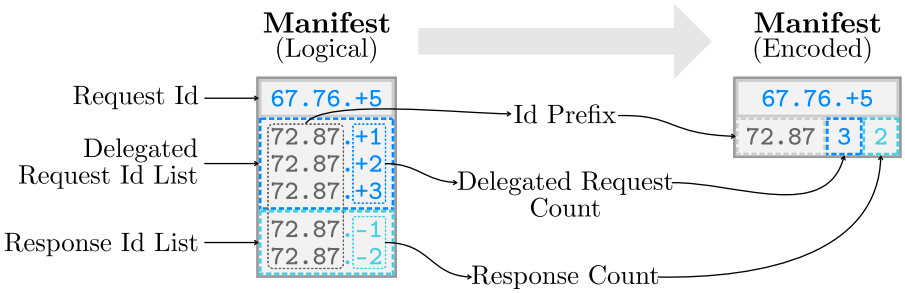


Figure 4.2: The contents of an example manifest shown in both its logical, variable-length format and its encoded, fixed-size format. Since all outgoing delegated request and response messages share the same id prefix (`72.87`), which includes the node id and the execution context number, the prefix only needs to be stored once in the encoded format. Since delegated request and response message numbers are assigned sequentially, the encoding can represent the two id lists using two counts: the count of delegated request messages and the count of response messages.

node id, execution context number, and message number; the node id and context number are shared for all messages sent as part of processing a single request. Instead of arbitrarily assigning message numbers, request messages are assigned sequentially increasing numbers starting at $+1$. Similarly, response messages are assigned sequentially decreasing numbers starting at -1. This assignment scheme relies on the fact that each request execution has its own range of message ids that it can assign. With this assignment scheme, the request and response message id lists can be encoded as the node id, context number, and the request and response message counts. Figure 4.2 shows the conversion from the naive manifest format to its fixed-size encoding. This encoding allows the manifest information to be represented in a small and fixed number of bytes, regardless of the degree of fan-out.

Next, the protocol is modified to piggyback manifests on existing messages, rather than using dedicated manifest messages. When a server finishes processing a request, it generates the request's manifest and piggybacks it on the last outbound message (this guarantees that the manifest is not prematurely generated). If the last message is a response, the manifest will be sent to the client. If the message is a delegated request, the manifest will be forwarded to another server who will then be responsible for piggybacking the manifest on one of its outbound messages. Forwarding ends when a manifest reaches a leaf in the request tree and all outbound messages are responses. In the special case where a leaf request sends zero application-level response messages, the manifest is piggybacked on a "null response."

The most obvious way to forward manifest information is for a server to add any manifest(s) it received from its parent to its own manifest, and forward all of them to its last child. However, this results in a variable number of manifests in the message to the last child and no manifest in the messages to other children. Instead, the manifests are distributed such that the parent's manifest goes in the first outbound message and its own manifest in the last outbound message. This ensures that no message has more than a single manifest, but it only works if a processed request generates two or more outbound messages.

When a server processing a request only sends a single outbound message, the protocol avoids the need to generate a new manifest for the request so that only the parent's manifest is forwarded. If the only outbound message from the server is a response, a flag is set in the response to indicate this fact along with the processed request's message id; the client then knows that there is no additional manifest information for the processed request. If the only outbound message from the server is a delegated request, the delegated request includes the parent request's message id so that the delegated request can inherit the parent request's identity; there is no need to generate a distinct manifest for the parent request since, from the client's standpoint, the two requests can be treated as one. Additionally, this optimization can be chained so that an identity is inherited by multiple generations when the topology consists of a branch of single delegated requests.

All manifests are eventually delivered back to the client without the use of separate manifest messages. In summary, the protocol can determine the completion of DPCs with arbitrary topology using a small and fixed amount of additional overhead per message.

4.3.2 Implications

The manifest mechanism has two key implications: one on the DPC server API and another on the timing of sending messages. First, because the manifest must contain the finalized id information for a request's generated responses and delegated requests, the protocol must know when the processing of a request will no longer generate more outbound messages. This is the reason for the `finish()` server API call.

Second, because the protocol needs to wait for the `finish()` call to generate the manifest and because the manifest is piggybacked on an outbound message, the sending of one outbound message must always be delayed until `finish()` is called. While the choice of which message to delay doesn't matter from the perspective of the protocol, a reasonable

scheme would be to delay each outbound message until the next outbound message is generated with the last message being delayed until the `finish()` call. This scheme would allow messages to be sent in order at the expense of causing some delay to outbound messages. An alternative API could include a flag in calls to send responses and delegated requests to indicate whether or not the outbound message is the last message. This additional flag would allow the protocol to send the messages immediately but would require the applications to know at the time of sending a message whether or not it will need to send more.

4.4 Failure Detection

The previously described completion detection mechanism allows a client to positively determine when a DPC execution has completed. However, if a DPC is not yet complete, can a client tell whether the DPC is still in progress or whether the DPC execution has encountered a failure? The purpose of failure detection is to allow a client to distinguish between an execution that can no longer make progress and an execution that is simply taking longer than expected to complete. While it is impossible to always correctly distinguish between the two states in a truly asynchronous environment [15], the goal for the protocol is to provide the client with an answer with a high degree of confidence. This is important as frequent false positives and negatives can adversely affect applications. For example, misidentifying an execution as failed can lead the application to unnecessarily retry the DPC and believing a DPC to be in progress when it has in fact failed can cause the application to wait indefinitely.

The protocol detects failures indirectly by trying to confirm whether a DPC can still make progress and assuming a failure has occurred if this cannot be confirmed within a *DPC timeout* period. Since a DPC is still in progress if all of its incomplete requests are in progress, the failure detection protocol works by checking the status of each incomplete request. To confirm whether a request is in progress, the client sends a ping to the server processing the request to verify two conditions: (1) the server is still alive and (2) the server is aware the request is in progress. If both conditions are confirmed for every incomplete request, the protocol assumes the DPC will continue to make progress and the DPC is considered in progress. However, if all incomplete requests cannot be confirmed to be in progress within the DPC timeout period, the DPC is considered failed.

Because the detection mechanism relies on timing, it is susceptible to false positives

in certain situations; the protocol tries to minimize the possibility of false positives but it cannot eliminate them entirely. The primary concern is when communication is lost or temporarily delayed; in such cases, the protocol may not be able to confirm a request is in progress, even though it is. This can occur if either the ping or its response is lost or delayed, or because the ping has arrived before the request has reached the server because the request was delayed. To reduce the likelihood that transient effects will cause false positives in failure detection, the protocol performs multiple rounds of pings before declaring a DPC has failed. If a ping fails, the ping is repeated after some delay. If a subsequent ping succeeds, the request is still considered in progress. Only after repeated ping failures does the protocol consider the request, and thus the DPC, a failure. However, if the issue persists longer than the DPC timeout, the protocol will incorrectly report a failure. Of course, if the DPC actually finishes before the timeout is reached, the client will already know it is a success and the failure detection mechanism will be halted.

To send pings for failure detection, the client needs to know all the incomplete requests and also which server is processing each request. However, the client initially knows neither. This is because DPC allows requests to be generated by intermediate servers without consulting the client. While the client might be aware of some requests from the manifests it has already received, it will not know which servers are responsible for those requests. Furthermore, since the client likely hasn't received all manifests, there may be incomplete requests that the client is completely unaware of. The protocol solves these issues using a number of mechanisms that are discussed in more detail in the following sections.

As explained in the following sections, the failure detection protocol is optimized for the common case and adds very little overhead to normal operation. In the most common case where a DPC completes in a timely manner, failure detection is not triggered and, supporting the protocol's various mechanisms only adds a single piece of metadata to manifests. When dealing with slow DPCs, the protocol work is proportional to the number of incomplete requests, which is normally quite small.

4.4.1 Server Discovery

When failure detection is initially triggered, the client does not have the server addresses needed to send pings for each incomplete request. While the client knows the destination servers for the requests it sent itself, it does not have this information for the delegated requests it discovered through the manifests it received. This is because manifests contain

request message ids but do not indicate which server is processing each request. To send the pings, the client must discover the destination server address for each request.

A simple but expensive solution is to include the destination server address for each delegated request in the manifest. The server that sends the delegated request knows the destination address and can cache this information to be included later when the manifest is generated. Unfortunately, adding a server address to the manifest for each delegated request causes the size of the manifests to be variable and potentially unbounded, something that the optimizations described in Section 4.3 were designed to prevent. Furthermore, this additional overhead would be incurred for all DPCs, even though this address information is only needed for slow DPCs that trigger the failure detection process.

To avoid paying the cost of collecting all the server addresses all the time, the protocol uses a two-step discovery process where the bulk of the work is only performed when an address is needed. In the first step, servers tag manifests with their own address. When a client receives a manifest and learns of a new delegated request, it won't know the request's destination server but it will know the source server that initiated the request. The only cost of the first step is the additional space in the manifest required to store the address. In step two, if pings are triggered and the client needs a request's destination address, the client can send a server discovery query to the request's source server. The source server would then respond with the request's destination server address. The client can then cache this information so this two-step process need only be performed once. This mechanism allows clients to discover a request's destination server on demand, incurring minimal overhead when the information isn't needed.

4.4.2 Request Discovery

The failure detection protocol relies on being able to verify the progress of every incomplete request in a DPC. Unfortunately, the client may not be aware of all these requests. Though the client will naturally discover some of a DPC's requests as manifests are received, the list of requests may be incomplete since some manifests may be missing when failure detection is triggered. Because of this, the failure detection protocol needs an on-demand mechanism for discovering all the in-progress requests.

Conceptually, the request discovery mechanism works by expanding the topology tree from the top down starting from the client's initial request(s): for each of the current leaves of the request tree, the mechanism queries the responsible server to learn if the

leaf request generated any delegated requests (i.e. the leaf's children in the request tree). Querying for a request's list of delegated request ids also returns the target server address for each delegated request; this allows the mechanism to continue walking the tree for newly discovered requests. This process continues until no new requests are discovered.

In the extreme case, the request discovery must walk the entire tree topology. Worse yet, the request discovery mechanism is necessary each time failure detection is triggered since the in-progress DPC could have introduced new requests to the execution. Fortunately, the request discovery mechanism can skip over any requests that are known to be complete. This is because completed request executions cannot introduce new delegated requests. If a previous query indicated that the request had completed, the request's previously declared list of delegated requests is final. If a request's manifest is received, the identified set of delegated requests is also final. In this way, the discovery mechanism only needs to process requests in the tree that are not known to be complete, significantly reducing the number of discovery queries necessary in many cases.

The request discovery mechanism has just one corner case to handle that relates to the use of inherited request ids. Inherited requests ids are used by the previously described manifest optimization that allows multiple requests in a branch of single delegated requests to share a single manifest. All requests in such a branch share the same inherited id and the resulting manifest is identified by this id. However, for request discovery and failure detection in general, each request must be uniquely identifiable and thus their intrinsic message id is used. When trying to skip request discovery for completed requests, the protocol will try to match a request with any received manifests to see if the request is already complete. When the request is not part of a branch of single delegated requests, the matching works as expected since the inherited id used in manifests is the same as the request's intrinsic message id. However, when the request does belong to such a branch, the ids will not match. To solve this, request discovery queries will also return a delegated request's inherited id, but only in the case where the delegated request is the only outbound message. Using the inherited id, the request discovery mechanisms can then match requests in a single delegated request branch to their shared manifest.

4.4.3 Ping-Pong Mechanism

While the status check, request discovery, and server discovery mechanisms are logically separate, they are combined into a single mechanism for efficiency. The failure detection

protocol uses a *ping* message, containing a request's message id, to initiate a status check of the request, to query for the request's list of delegated requests, and also to collect the target server address for each delegated request. In response to a ping, the server will reply with a *pong* if the request being pinged is known to the server. Otherwise, it is ok for the server to not respond as the protocol does not need to distinguish between a dead server and a live server with no knowledge of the pinged request. The pong includes a flag indicating whether or not the pinged request has completed; this allows the client to stop pinging completed requests. If a client does not receive a pong, it considers the status check to have failed since either the server has crashed or the server did not respond because the request is not known to the server. The pong also includes the pinged request's most up-to-date list of all the delegated requests and their destination addresses. This allows the client to discover both new requests and destination addresses when pings start to occur.

The failure detection mechanism is automatically triggered on incomplete DPCs periodically with a round of pings being sent every *ping period*. Each round, a ping is sent for each incomplete request. Not receiving an expected pong is a red flag but does not immediately signify a failure. Instead, the protocol waits for multiple rounds of pings to see if a pong is received for any round. If a client does not receive at least one pong for each request within a *DPC timeout* period, the DPC is declared a failure. Otherwise, the timeout is reset and the DPC is considered still in-progress until failure detection is triggered again.

The best choice of ping period and DPC timeout value will depend on the application and execution environment. The ping period controls not only the frequency of pings but also the time a client will wait before triggering its first round of pings. Because triggering pings adds overhead, DPCs are most efficient if they complete within one ping period, before any pings are triggered. For this reason, the ping period should be set large enough to cover most DPC executions. However, setting a larger ping period also affects how quickly failure detection can report a failure. This is because the DPC timeout must be set large enough for multiple rounds of pings to complete and thus must be larger than the ping period. Additionally, larger DPC timeout values affect how much server-side state will accumulate; this is discussed in more detail in Section 4.4.4. For most applications, it should be possible to find a ping period and timeout value suitable for all the application's DPC executions. However, it is also possible to set these values per execution as the protocol only assumes these values are constant within a DPC execution.

4.4.4 Garbage Collection

In order for servers to properly respond to pings, servers must ensure information like a request's list of delegated requests and their destination addresses is kept around until the client no longer needs the information. This is done using a simple timeout mechanism where the information is kept around until the timeout elapses. The timeout, called the *gc timeout*, starts when a request finishes processing and is reset when the request is pinged, allowing pings to act like a "keep-alive" for the server-side state. When the gc timeout elapses, the server knows the client has not pinged in some time and the information can be garbage collected. Since requests that have been confirmed complete will not be pinged, most of a DPC's server-side state can be cleaned-up even when a DPC has many hops or is delayed by stragglers.

The gc timeout must be set large enough so that metadata is kept around as long as a ping might be received. One way to set this value is relative to the DPC timeout. Since the client will consider a DPC failed if no pongs are received within a DPC timeout, the server knows that if no pings are received for longer than the DPC timeout, the client is either no longer sending pings to this server or it will soon declare the DPC failed because this server has not sent any pongs. In either case, metadata is no longer needed after a DPC timeout period has elapsed with no pings. Logically, setting the gc timeout to be as large as the DPC timeout should be sufficient. However, if the different machines have different clock rates, it is possible for the gc timeout to expire on one machine before the DPC timeout elapses on another; this creates an opportunity for false-positives in the failure detection mechanism. As such, the gc timeout should be set comfortably larger than the DPC timeout to account for any potential differences in clock rates. In most applications, universal settings for the ping period, DPC timeout, and gc timeout can be used. However, if the client chooses different DPC timeout values for each DPC execution, this must be communicated to the server so that the gc timeout can be set accordingly; the timeout setting could be included in each request.

Optional Active Signaling

The amount of space a server needs to temporarily store failure detection metadata is a function of the application's throughput and the configured gc timeout. For example, with 100,000 requests per second and a 10ms gc timeout, a server would need space to store 1000

requests worth of metadata in the steady state; this adds up to a few hundred kilobytes of total space. This level of server-side state should be fine for most applications. Even if the application's request rate and timeout period were increased by 10x, the megabytes of space needed would likely still be workable in data center deployments.

If the amount of server-side state is still a concern, the protocol can be modified to reduce the running amount of metadata stored by triggering early garbage collection using an additional active signal. The basic idea is that the client can signal that it is safe to garbage collect a particular DPC once a DPC is declared complete. Because most DPCs are expected to complete well before the gc timeout, the signal can be delivered early enough to shorten the lifetime of metadata data and thus decrease the overall metadata footprint.

There are a number of ways this active garbage collection signal can be delivered to servers. The most obvious but high overhead approach is to propagate garbage collection messages along a DPC's request tree. When a DPC completes, the client would send messages to the servers that it is aware of to signal that the metadata for a particular DPC can be garbage collected. Those servers would then forward the message to any servers they delegated requests to; this repeats until all the servers involved in the DPC have received the signal. This approach allows the signal to reach all servers involved relatively quickly and thus allows all associated metadata to be garbage collected early. However, it also adds another message to be processed for each request in a DPC, a pretty significant overhead.

Because garbage collection signals are completely optional from the protocol's perspective, a much more opportunistic approach could provide some of the benefits of early garbage collection while being relatively efficient. Rather than generating signals for each DPC completion, a client can keep track of its oldest in-progress DPC. Assuming DPC ids are assigned sequentially, the id of the oldest in-progress DPC can be used to signal that all older DPCs are no longer in progress and can be garbage collected. The oldest in-progress DPC id can then be included when a client sends a request and also be forwarded in any resulting delegated requests. This mechanism adds a small number of bytes to request messages but has otherwise no additional overhead. The drawback is that not all relevant servers are guaranteed to receive the garbage collection signal. Still, in complex distributed applications, it seems likely that servers will be involved in many DPCs and thus will at least periodically receive these signals. This is just one simple way to disseminate a garbage collection signal; more sophisticated schemes can be developed if the need for early garbage collection is particularly critical.

4.4.5 Handling Other False Positives and Negatives

The failure detection protocol tries the minimize the likely-hood it will incorrectly distinguish between a failed and in-progress DPC in most real world situations. However, there are two additional cases that would require protocol modifications to address.

First, a server crashing *after* it finishes processing a request may result in an erroneous failure designation. In this case, the processing can and does continue since any delegated requests and responses would have already been sent. Unfortunately, the crashed server would not be able to answer pings. If the DPC doesn't complete before the DPC timeout expires, the unanswered pings would eventually be interpreted as a DPC failure, even though the DPC may actually still be executing. There are likely augmentations to the protocol that allow peer servers to detect the server crash and proactively send the crashed server's manifest back to the client to mitigate this corner case but this only needs to be explored if this case proves to be a common occurrence.

Second, it is possible that a DPC will no longer make progress even though all servers are alive and all requests were received; this would lead failure detection to believe the DPC is still in progress when in reality it should be considered a failure. For example, if an application-level issue prevents a DPC from making forward progress (e.g. livelock, starvation, etc), the DPC will never complete even though the server itself is alive and can confirm, at least at the DPC level, that the request is in progress. To handle these situations, additional checks are needed to confirm a request is indeed in progress. For example, the status check could include application-level health checks to ensure that progress has not stalled in the layers above the DPC communication layer. Checks like these would require additional integration but would be important if these complex failure modes are common.

4.5 Execution Abort

Aborting the execution of an in-progress DPC is not included as part of the basic DPC protocol since not all applications need this ability. However, the basic protocol can be easily extended to support this feature by leveraging the mechanisms already in place to tackle failure detection.

From the perspective of an application, aborting a DPC is an out-of-band signal initiated by the client that causes the DPC's in-progress request executions to terminate. The challenge for the protocol is identifying and locating those in-progress servers to deliver the

abort signal; only in-progress servers need to be signaled since a completed request has no execution to terminate. Once those servers are identified, it is simply a matter of sending the signal and propagating it up to the application.

The core challenge for the abort protocol is essentially the same as that of failure detection; the client initially knows neither the full set of in-progress requests nor the servers that are processing the requests. Thus, the mechanisms used for failure detection can also be used to support the abort protocol. The ping-pong mechanism used in failure detection allows the client to discover a DPC's set of in-progress requests and will send a ping message to each server involved in the process. When the client initiates the abort, the ping-pong protocol can be triggered with an abort signal piggybacked along with each ping. After a server has processed the abort signal and can ensure that no additional delegated requests will be sent, the server's pongs can be marked to indicate the abort is complete. As with failure detection, completing the abort protocol may take multiple rounds of ping-pongs. Once all servers have reported that their aborts have completed or have otherwise finished processing their requests, the client can be sure that all processing has stopped and no new requests will be generated.

The general design of the abort protocol allows applications to handle the abort signal as they see fit, however, the protocol can be modified to forcibly curtail DPC processing. Instead of waiting for applications to process the abort signal and indicate they have aborted, the DPC layer can ignore API calls to send delegated requests after receiving the abort signal and immediately mark the request aborted. While the processing of the in-progress requests may continue, the DPC cannot grow since new delegated requests cannot be sent.

4.6 The Transport Below

The DPC protocol is logically described in terms of sending and receiving messages and thus can be implemented on top of any network transport layer providing point-to-point communication. While it would be possible to implement DPC in ways that subsume the responsibility of traditional network transports, in practice, it makes more sense to rely on existing solutions for congestion control, error detection, and potentially some forms of reliability in the face of packet loss. For example, a DPC layer could be implemented on top of TCP by mapping DPC messages onto TCP streams.

While most aspects of mapping DPC messages onto various network transports should be

relatively straightforward, there is one requirement that may require some additional care. The DPC protocol as described assumes that messages are sent with at-most-once semantics; when the protocol sends a message it expects the message to be delivered either once or not at all. Depending on whether the underlying transport already provides this guarantee, additional mechanisms may need to be added to implement at-most-once semantics.

If an underlying transport already provides at-most-once semantics, no additional mechanism is needed. For example, TCP provides at-most-once semantics. If a TCP connection is closed before a message's bytes have been acknowledged there is no way of knowing whether the message was delivered. However, TCP does guarantee that no duplicate bytes will be delivered and thus any message sent over TCP is delivered at most once. As such, no additional mechanism is needed; the DPC implementation must simply ensure that the DPC layer itself will never try to resend a message on a different connection when a connection closes unexpectedly.

If an underlying transport does not guarantee at-most-once semantics, additional mechanisms will be needed to filter out duplicate messages. For example, UDP packets can be duplicated at the network level, potentially causing redelivery of messages. To handle this, receivers would need to keep around enough state to know which messages have already been received and filter out any redelivery. Notably, this mechanism would also need to work across server crashes so that a message delivered before a crash will not be accepted if redelivered after the crash. One common way to solve this would be to use some form of session management where messages are sent as part of an established session and are only accepted when the session is active. While implementing DPC on top of transports like UDP requires a bit more mechanism, it shouldn't be a significant technical challenge.

4.7 Summary

The DPC protocol solves the main challenges of using multi-hop communication, primarily addressing how a client can detect DPC completion or failure when the execution topology is initially unknown. The protocol involves techniques that allow the client to discover the necessary topology information in an efficient manner and uses this information to facilitate the DPC abstraction and API. The protocol itself is complex and hints at the difficulty that an application developer faces when attempting to implement multi-hop executions in an ad hoc manner. For DPC, however, the protocol is intended to be implemented as a library

and hidden by the abstraction. Ultimately, this makes it easier for developers to build multi-hop applications.

Chapter 5

The Details of Implementation

In developing the DPC protocol, I also built a DPC reference implementation. This implementation was useful in validating and iterating the design of the DPC protocol and was ultimately also used to evaluate DPC's performance characteristics. The implementation took the form of a C++ user-space library called *Roo*. Roo implements the DPC API and core protocol as previously described with only slight modifications to the API to better fit C++'s object-oriented nature. The source code is freely available [25].

Following the DPC protocol, Roo only implements the DPC layer and relies on a separate network transport layer to provide point-to-point message delivery. While Roo could have been implemented on top of any network transport, I chose to build and use a new network transport implementation, based on the Homa [30] transport protocol, that focuses on providing low-latency message-oriented communication in a data center environment. This Homa transport implementation is also a user-space library and relies on DPDK [4] to send and receive raw Ethernet packets while bypassing the kernel networking stack. While used in Roo, the Homa library is general enough to be used in any application requiring message-oriented communication. The source code for the library is also freely available [26].

The remainder of this chapter focuses on Roo and in particular the aspects of the implementation that may be relevant to developers who want to build other DPC implementations in the future.

5.1 Object-oriented API

As a library implemented for C++, Roo reinterprets the DPC API in a more object-oriented style. Roo's DPC API is structured around a handful of classes; each API call described in Section 3.2 is reinterpreted as a method on one of Roo's API classes. Roo also introduces a few additional concepts that address various practical constraints specific to Roo's chosen target environment. For example, Roo does not use threading internally and has no dependencies on any particular threading architecture. Instead, it relies on applications to invoke its APIs to make progress on background tasks. Additionally, Roo's API is non-blocking since blocking is also a threading-specific feature. These designs allow applications of Roo to use any threading architecture they wish.

5.1.1 `Socket`

Applications using Roo will first encounter the `Socket` class. Where the DPC protocol is described as if each machine is a single communication end-point, Roo uses a `Socket` to represents a logical DPC communication end-point. Typically, each application client or server only needs a single `Socket` because one `Socket` can be used simultaneously to communicate with many peers. Unlike a TCP socket, which serves as a point-to-point connection, a Roo `Socket` can send and receive DPC messages to and from any other Roo `Socket`.

Some deployments will need multiple logical end-points on a single machine. For example, there may be a need for multiple applications per machine with an end-point for each application or perhaps even multiple end-points per application. To support this, each `Socket` acts as a logically independent DPC client and/or server node; they can be individually addressed and are each assigned a unique node id as described in Section 4.1.2.

While Roo could have been designed with separate client and server socket types, Roo's `Socket` implementation supports both client and server use so that a single socket can serve either or both purposes. To that end, the `Socket` has two main methods (shown in Figure 5.1). First, the `allocRooPC()` method is used by clients to prepare a new DPC. Second, the `receive()` method is used by servers to receive incoming DPC requests. Additionally, both methods return the objects as Roo-wrapped "smart pointers" (`Roo::unique_ptr`). In addition to helping support Roo's memory management scheme (discussed in section 5.6), the smart pointers allow Roo to trigger special "cleanup" logic when a pointer goes out of

```
/**
 * An end-point for RooPC client and server communication.
 */
class Socket {
  public:
    /**
     * Allocate a new outgoing RooPC associated with this socket.
     */
    virtual Roo::unique_ptr<RooPC> allocRooPC() = 0;

    /**
     * Check for and return an incoming request.
     *
     * @return
     *      An incoming request, if available; otherwise, an empty pointer.
     */
    virtual Roo::unique_ptr<Request> receive() = 0;

    /**
     * Make incremental progress performing Socket management.
     *
     * This method MUST be called for the Socket to make progress and should
     * be called frequently to ensure timely progress.
     */
    virtual void poll() = 0;
};
```

Figure 5.1: A `Socket` object represents a Roo end-point and is part of both the client and server APIs.

scope. Both `allocRooPC()` and `receive()` will be discussed in more detail in Section 5.1.3 and Section 5.1.4 respectively.

In addition to these two methods, the `Socket` also has a `poll()` method. The `poll()` method must be called by the application periodically so that the Roo library can execute background DPC protocol logic. Roo does not use threading internally and thus has no dedicated thread for background processing. As a result, Roo must rely on application threads to call `poll()` to give Roo a chance to perform background work. The background work performed in the call includes checking for incoming messages and performing protocol processing on them. Additionally, the various protocol timers and timeouts are also managed within this background logic because the Roo library has no thread with which to receive interrupts. If an application doesn't want to worry about calling `poll()`, it could simply create a dedicated thread that calls `poll()` in an infinite loop.

5.1.2 `InMessage`

In the Roo API, incoming messages are represented as `InMessage` objects. An `InMessage` provides methods for accessing the message's contents and getting its length (shown in Figure 5.2). Incoming messages are either response messages received by the client or request messages received by the server and so `InMessage` is used as the return type for the related client and server `receive()` API calls.

Giving applications access to incoming messages using the `InMessage` abstraction allows applications to incrementally decide on how to copy out the contents of the message. In contrast, many C/C++ network transport APIs accept a buffer into which the entire message is copied. This presents a challenge since an application may not know where the message contents should be copied until it has processed at least part of the message. For example, a message may need to be deserialized into various data structures but the application may not know how to perform the deserialization until after it processes the message header. If the entire message was first copied into a buffer, parts of the message might then need to be recopied into their final locations. By using the `InMessage` methods instead, an application can eliminate the intermediate copy and only copy each part once when its final location is known.

```
/**
 * Represents an array of bytes that has been received over the network.
 */
class InMessage {
  public:
    /**
     * Get the contents of a specified range of bytes in the InMessage by
     * copying them into the provided destination memory region.
     *
     * @param offset
     *      The number of bytes in the InMessage preceding the range of
     *      bytes being requested.
     * @param destination
     *      The pointer to the memory region into which the requested byte
     *      range will be copied. The caller must ensure that the buffer is
     *      big enough to hold the requested number of bytes.
     * @param count
     *      The number of bytes being requested.
     *
     * @return
     *      The number of bytes actually copied out. This number may be less
     *      than "count" if the requested byte range exceeds the range of
     *      bytes in the InMessage.
     */
    virtual std::size_t get(std::size_t offset, void* destination,
                            std::size_t count) const = 0;

    /**
     * Return the number of bytes this InMessage contains.
     */
    virtual std::size_t length() const = 0;
};
```

Figure 5.2: The `InMessage` API gives applications access to the contents of an incoming message.

```
/**
 * A Roo DPC handle representing a collection of requests and an associated
 * collection of responses that can be sent and received asynchronously.
 */
class RooPC {
  public:
    /**
     * Send a new request for this RooPC asynchronously.
     *
     * @param destination
     *      The network address to which the request will be sent.
     * @param request
     *      First byte of a buffer containing the request message.
     * @param length
     *      Number of bytes in the request message.
     */
    virtual void send(Address destination, const void* request,
                      std::size_t length) = 0;

    /**
     * Return a received response for this RooPC.
     *
     * @return
     *      Returns an incoming response message, if available; otherwise,
     *      a nullptr is returned. Ownership of the returned message is NOT
     *      passed to the caller; the lifetime of the returned message is
     *      tied to this RooPC
     */
    virtual InMessage* receive() = 0;

    /**
     * Check and return the current Status of this RooPC.
     */
    virtual Status checkStatus() = 0;

    /**
     * Wait until all expected responses have been received or the RooPC
     * encountered some kind of failure.
     */
    virtual void wait() = 0;
};
```

Figure 5.3: The `RooPC` abstract class represents a DPC in Roo's client API.

5.1.3 `RooPC`

In the API described in Section 3.2, client API calls were associated with a specific DPC handle. In Roo, the API calls are methods on a `RooPC` object, an object that represents a DPC. Whenever an application client wishes to initiate a new DPC, it first calls `Socket::allocRooPC()` to create a new `RooPC` object; the object is returned as a Roo smart pointer. The `RooPC` is logically associated with the `Socket` that created it and all the `RooPC`'s responses use the `Socket` as their destination end-point. With a `RooPC` object, applications can call the `send()`, `receive()`, and `status()` methods (shown in Figure 5.3) that map directly to the DPC API calls of the same name. When the application client is finished with the DPC, it can simply allow the smart pointer to go out of scope, triggering Roo's internal memory management.

In addition to the core DPC client API calls, Roo adds the `wait()` method to `RooPC` objects. The `wait()` call is a convenience method that allows applications to "block" waiting for a DPC to complete. Because Roo makes no assumptions about an application's choice of threading and thus cannot use any blocking primitives, Roo implements all of the DPC API calls in a non-blocking manner. This places a burden on applications to loop, checking on the status of a `RooPC`. Some applications will take advantage of the non-blocking calls. For example, applications could monitor the activity of multiple DPCs concurrently by looping through and checking each incomplete DPC. However, if an application does want to wait for a single DPC to complete, it can call `wait()`. The `wait()` implementation itself also does not use any blocking primitives. Instead, it repeatedly calls `poll()` and checks on the DPC's status in a tight loop, only breaking the loop to return from the `wait()` call when the DPC is no longer in progress.

5.1.4 `Request`

Like the client API, the server API described in Section 3.2 uses a request handle to associate API calls with specific requests executions. Again with Roo, the API calls are methods on a `Request` object, representing an incoming DPC request. Application servers can receive a `Request` through the `Socket::receive()` call. The `Request` is a subclass of `InMessage` so applications can access the contents of the request by using the `InMessage` method calls. The other DPC server API calls, `reply()` and `delegate()`, are also methods on a `Request` (shown in Figure 5.4).

```
/**
 * A handle for an incoming request providing access to the request message
 * and an interface for sending a response or additional requests.
 */
class Request : public InMessage {
  public:
    /**
     * Send a message back to the initial RooPC requestor.
     *
     * @param response
     *      First byte of a buffer containing the response message.
     * @param length
     *      Number of bytes in the response message.
     */
    virtual void reply(const void* response, std::size_t length) = 0;

    /**
     * Send a message as an additional request for the associated RooPC.
     *
     * @param destination
     *      Address to which the new request message should be sent.
     * @param request
     *      First byte of a buffer containing the request message.
     * @param length
     *      Number of bytes in the request message.
     */
    virtual void delegate(Address destination, const void* request,
                          std::size_t length) = 0;
};
```

Figure 5.4: The `Request` is part of Roo's server API and acts as the server-side handle for the execution of an incoming request.

Notably absent is the `finish()` DPC server API call (seen in Section 3.2). Because a `Request` is provided to the application server as a Roo smart pointer, Roo can trigger cleanup logic when a `Request` goes out of scope. This cleanup logic includes the protocol work that would have been triggered by a `finish()` call.

5.2 Decomposition

Each aspect of the DPC protocol can be split into the client-side logic and the server-side logic. This split is also naturally reflected in the implementation. The client-side logic is implemented within the `RooPC` class, whereas the server-side logic is implemented within the `Request` class.

The DPC protocol is naturally event-driven and so the logic within the `RooPC` and `Request` classes are organized to trigger on their corresponding events. There are three main kinds of events. First are the API methods that trigger outbound messages. The calls to `RooPC::send()`, `Request::delegate()`, and `Request::reply()` update the tracking information used for completion detection and failure detection whereas the implicit call to "`finish()`" triggered by destruction of a `Request` smart pointer completes the generation of the request's manifest. Second, the protocol processes incoming messages as they arrive; this includes both application-level messages (i.e. requests and responses) as well as protocol-level messages (i.e. pings and pongs). For example, when a response is received, a `RooPC` will need to update the completion detection tracking information with the receipt of the response and any piggybacked manifest. Third, the protocol is also triggered on timer events. The triggered logic executes the various aspects of failure detection including the sending of pings and the decision to garbage collect server-side state. All protocol logic is triggered in one of these three ways.

In addition to the DPC protocol logic itself, Roo must also provide a means of receiving incoming messages and generating timer events. This support is implemented in the `Socket` class. Because Roo does not internally use any threads, it relies on applications to periodically call `Socket::poll()` to perform background work. This includes receiving incoming messages from the underlying Homa transport and dispatching it to the appropriate `RooPC` or `Request` object. If the incoming message is a response, the message is dispatched to and processed by the associated `RooPC` object. If the incoming message is a request, a new `Request` object is constructed with the request message as input. Similarly,

ping and pong messages are also dispatched to the appropriate `Request` and `RooPC` objects. The `Socket::poll()` call is also responsible for triggering timer events. Conceptually, each `RooPC` and `Request` object registers a timer handler with a `Socket` object and specifies the time after which the handler should fire. When `Socket::poll()` is called, the execution checks whether any timers have elapsed and fires the handlers accordingly.

5.3 Request and Response Identifiers

Having been developed before the DPC protocol was formalized, Roo retains some artifacts of this history. One artifact is the way request and response ids are specified. In the formal protocol described in Section 4.1.2, all messages use the same id format. However, in the Roo implementation request and response ids use different formats. This is possible because the ids of requests and responses are never intermixed and are contextually separated. This separation led to the format of the ids being specialized for their own use cases. None of these specializations are strictly necessary. In the best case, they only serve to save a few bytes on the wire; in the worst case, they increase the complexity of implementing the protocol logic. For this reason, I don't necessarily recommend the use of these formats, rather I describe them here for completeness.

At its core, both request and response ids share the same three-part id scheme described in Section 4.1.2. This includes the globally unique node id, an execution context number, and a message number. However, since request and response ids are never intermixed, the message number space does not need to be separated into positively and negatively signed numbers for requests and responses respectively. Instead, Roo simply numbers both message types with unsigned numbers. Though this can cause a request and response to have the same id triple, they are used for different purposes and will never result in misidentification.

Roo further specializes the format of request ids to save a few bytes on the wire. Because of the DPC protocol's optimizations to eliminate manifests from request executions that only generate a single delegated request, requests must logically carry two request ids: the request's own unique message id, and the message id it inherited from its parent request. The request's own unique id allows the failure detection protocol to identify and verify the status of individual requests, while the request's inherited id allows the completion detection protocol to treat all requests within a branch of single delegated requests as a single request. To save a few bytes, Roo merges the functionality of both ids into a single

four-part id consisting of the basic id triple plus an additional sequence number. When request ids are being generated and there is more than one outbound request, the basic triple is generated as before and the extra sequence number is filled with a zero. However, when only a single delegated request is generated, the delegated request is assigned an id based on the parent request's id. The delegated request adopts the parent request's basic triple and increments the parent request's sequence number. In this way, completion detection can use just the basic triple and will treat all requests within a branch of single message delegations as a single request, as expected. However, for failure detection, the full four-part id is used to uniquely identify each of the requests within a branch. For these reasons, I will later refer to the basic triple as a request's *branch id*. Again, this saves some bytes on the wire but does increase the implementation complexity.

I also considered changing the way response ids were generated to save a few additional bytes on the wire, however, I ultimately decided against implementing this optimization. Manifests and responses must already carry the id of the associated request, so generating response ids based on the associated request id could save the implementation from having to use a separate node id and execution context number (the first two parts of the basic id triple). Instead, a response id would consist of the associated request's id followed by the message number as before. Using this scheme, manifests could be encoded as before but decoding response ids would use the associated request's id rather than the node id and context number. In a response message itself, its id would be encoded as just the message number since the prefix (i.e. the associated request's id) is already included for other completion detection purposes. Ultimately, this allows the response header to drop the bytes used to carry a node id and context number. However, this savings is likely not worth the additional complexity.

5.4 Data Structures

The data structures needed to implement the DPC protocol are all relatively simple. For completion detection (Section 4.3), the server must keep track of enough information to form manifests and the client must collect the topology information from these manifests and incoming responses. On the server side, no special data structures are needed in a `Request` object to collect the information needed in manifests; each of the fields of a manifest (i.e. the request's id, the node id, context number, and request and response counts) can be

tracked as individual variables until they are assembled into a full manifest. On the client, a `RooPC` tracks both the set of expected responses and the set of expected manifests. Since the protocol only needs to check that all responses and all manifests are received, the actual topology of the DPC doesn't need to be reconstructed. Instead, a simple map from id to receipt status is sufficient; a response is identified by its response id, and a request's manifest is identified by the request's branch id. When new responses and manifests are expected, entries are added to their respective maps. When either a response or a manifest is received, the map entries are marked accordingly. As a performance optimization, a separate set of counters is used to keep track of the number of missing requests and number of missing manifests so that completion checks can avoid scanning the map data structures and can instead simply look at the counts.

For failure detection (section 4.4), a server must remember the destination address for each delegated request and the client must track the status of known requests. On the server side, a `Request` object uses a simple vector to track the destination addresses; the vector is indexed by a request's message number. On the client, a `RooPC` must keep track of the known requests, their completion status, and the request id and server address to ping for status checks, request discovery, and server discovery (recall that the request id and server address to be pinged can either be for the request itself or the parent of the request). Like with completion detection, a simple map from request id to this metadata is sufficient.

Logically, the client-side expected-manifests map, used for completion detection, and known-requests map, used for failure detection, serve different purposes. However, Roo combines the two maps into one using the previously specialized request id scheme. The data structure maps from a request's branch id to the metadata needed for both completion detection and failure detection. When requests do not belong to the same branch, the map works for both use cases as before, since the branch uniquely identifies the request. However, when multiple requests belong to the same branch, they are only distinguished by their extra sequence number. In this case, the requests will all map to the same entry in the data structure since they share the same branch id. Fortunately, failure detection only needs to track a single request per branch at any given time. If two requests within the same branch are discovered only the request with the larger sequence number needs to be tracked since its existence implies its ancestor requests with lower sequence numbers must have already completed and no longer need to be tracked. While the merging of these two maps may be a slight performance optimization as the implementation can update both completion

detection and failure detection information with a single map lookup, it's not clear that marginal benefits merit the increase in complexity. For this reason, I don't recommend this optimization for other implementations.

5.5 Thread-safety

Roo is designed with multi-threaded applications in mind and uses synchronization to ensure thread-safe execution where appropriate. Roo expects that multiple application threads might share a single `Socket` concurrently. Thus, Roo provides thread-safety for all `Socket` methods, primarily protecting against concurrent access to the data structures that track the `RooPC` and `Request` objects. This protection takes the form of a monitor-style lock on a `Socket` object.

On the other hand, Roo does not expect multiple threads to call methods on a single `RooPC` or `Request` object concurrently. However, the background work triggered by calls to `Socket::poll()` may also need to update `RooPC` or `Request` object state as incoming messages are processed or timer logic is triggered. For this reason, background work is handled by internally invoking methods on the related object. For example, `Socket::poll()` internally calls a method on `RooPC` to process an incoming response message. Following this design, the necessary thread-safety is provided by protecting both public API methods and internal methods against concurrent execution using a monitor-style lock for each `RooPC` or `Request` object.

Roo's synchronization primitive of choice is a simple spinlock implemented using atomic operations over a C++ `std::atomic_flag`. Spinlocks are used instead of locks based on kernel-provided synchronization primitives for two reasons. First, critical sections in Roo are very short; most span less than a few hundred nanoseconds of execution. The cost of suspending and resuming a thread using a kernel synchronization primitive is greater than the duration of the critical section; no other work could actually be done in the meantime and the thread would actually be blocked longer than if a spinlock were used. Secondly, Roo made the choice not to make any specific assumptions about the application's threading environment, including whether or not kernel threads are being used.

5.6 Memory Management

Roo performs all memory management internally and only exposes objects as smart pointers, as previously described; this minimizes the complexity of its API. The managed memory is used to hold both the metadata needed for the DPC protocol and the buffers used to hold the contents of messages; the metadata and buffers are associated with `RooPC` and `Request` objects. In many cases, Roo will require access to these objects both before and after they are used by applications. For example, while an application thread may no longer need a `Request` object because application-level processing is complete, Roo may continue to operate on the object internally to process concurrent or future pings and timer events. By managing memory internally, Roo can ensure that objects live long enough to serve both application processing and background DPC protocol logic. This frees applications from having to consider Roo's object lifetime requirements.

The smart pointers allow the application to easily express its expectations about whether the underlying object needs to be accessible by simply holding on to the pointer. Once the smart pointer is destroyed, Roo knows that the application no longer needs access to the object and is free to delete the object once Roo itself no longer needs it. Roo uses smart pointer for both `RooPC` and `Request` objects. This allows applications to exclusively worry about their own object lifetime requirements without worrying about Roo's internal needs. The only constraint Roo places on applications is that `Socket` objects must be held until all associated `RooPC` and `Request` objects are released, though even this requirement can be lifted with a bit more engineering effort.

For client calls to `RooPC::recieve()`, Roo returns a raw `InMessage` pointer instead of a smart one. This is because the lifetime of these pointers can be tied to the associated `RooPC` object. For example, so long as a client holds on to a `RooPC` smart pointer, all `InMessage` pointers returned via calls to the `RooPC`'s `receive()` method will remain valid. Once the `RooPC` smart pointer is released, so too can the associated `InMessage` objects.

Roo's memory management responsibilities extend to the buffers used to store inbound and outbound messages as well. In both cases, Roo manages the buffers because the lifetimes must extend beyond their usefulness to applications. Inbound message buffers are associated with `InMessage` objects and are allocated when messages arrive. This allows the message to be processed through the DPC protocol and held by Roo until it is requested by an application. Outbound message buffers are associated with the `RooPC` or `Request` object

whose method was called to send the message and must hold the contents of the outbound message until the message is completely sent. Since the Roo API is non-blocking, the `send()` calls may return before the message is actually sent. Additionally, because of manifest piggybacking, as described in Section 4.3, calling `reply()` and `delegate()` will not immediately send a message; messages are deferred until the next call to either method or until `Request` processing has finished. Holding a copy of the outbound message contents in a buffer obviates the need for applications to ensure the original buffer is valid past the invocation of the API call.

5.7 Summary

The vast majority of Roo's implementation is relatively straightforward following the API and protocol as described in Chapter 3 and Chapter 4 respectively. The implementation doesn't require any novel or complex data structures nor does it need any particularly exotic language features. The most challenging aspects of the implementation surround the aspects of managing memory and ensuring thread safety. However, these challenges are arguably common to all multi-threaded C++ code. Based on my experience with Roo, it seems plausible that the DPC API and protocol could likely be implemented in a variety of languages and environments.

Chapter 6

Evaluation

The main reason to adopt DPC and the multi-hop style of communication is to improve performance. Compared to using request-response style primitives like RPC, the multi-hop pattern can reduce communication latency by eliminating intermediate network-hops and messages. However, the complexity of DPC's underlying protocol also incurs a cost. This chapter evaluates the costs and benefits of using DPC vs RPC and compares the performance of a DPC library against the performance of a comparable RPC library in a variety of execution patterns. The aim is to answer the following general questions:

- Under what conditions will DPC provide lower latency than RPC?
- How much latency reduction can be expected?
- Does the use of DPC affect throughput and resource utilization?

6.1 Benchmark Setup

The DPC and RPC libraries used in this evaluation are built to share as much code as possible; this eliminates factors that are not germane to the comparison between DPC and RPC. The DPC library, Roo, as described in Chapter 5, is implemented in C++ on top of a DPDK-based [4], kernel-bypass, network transport implementation. The RPC library, also written in C++, implements a similar non-blocking API, is built on the same network transport, and includes the same threading and runtime code.

While much of the RPC library code and Roo code is the same, there are a number of places where the differences between the RPC and DPC abstraction allow the RPC

CPU	Intel E5-2640v4 10-core/20-thread at 2.4 GHz
RAM	4x 16 GB DDR4-2400 DIMMs
NIC	Mellanox ConnectX-4 25Gbps NIC
Switch	25Gbps Mellanox 2410 (Leaf) 5x100Gbps Mellanox 2700 (Spine)

Table 6.1: Experiments were run on the CloudLab [14] xl170 cluster which has the above hardware specifications.

implementation to be simplified and optimized. For example, RPC's logic for detecting RPC completion (i.e. has a response been received yet) is trivial compared to what is necessary for Roo.

The benchmarks are designed to expose any performance costs associated with the DPC implementation. Non-protocol-related costs are minimized or eliminated so that the protocol overhead will not be lost in the noise of other inefficiencies. Using a reasonably high-performance test cluster with fast networking (detailed in Table 6.1) and a kernel-bypass network transport designed for low latency minimizes the cost of networking and packet processing. Additionally, the benchmarks do not implement any application/service logic so that the measurements can focus on the cost of communication.

Unless otherwise specified, benchmarks were performed at low load (10kops/sec) using a single client machine and 20 server machines with each request sent to and processed by a randomly chosen server.

6.2 Basic Performance

There are two primary parameters that affect the relative performance of DPC and RPC. The first parameter is the number of nodes that must be visited in sequence (i.e. the number of "hops"). Figure 6.1 shows the execution latency as a function of the number of hops when using either DPC or RPC. This execution pattern is applicable to a number of applications including a proxy, a load balancer, the Chord lookup protocol [34], or Chain Replication [35]. In this test, the DPC-based execution uses delegated requests to directly communicate from server to server whereas the RPC-based execution issues a number of RPC calls in sequence.

The relative performance improvement of DPC increases with the number of hops. At

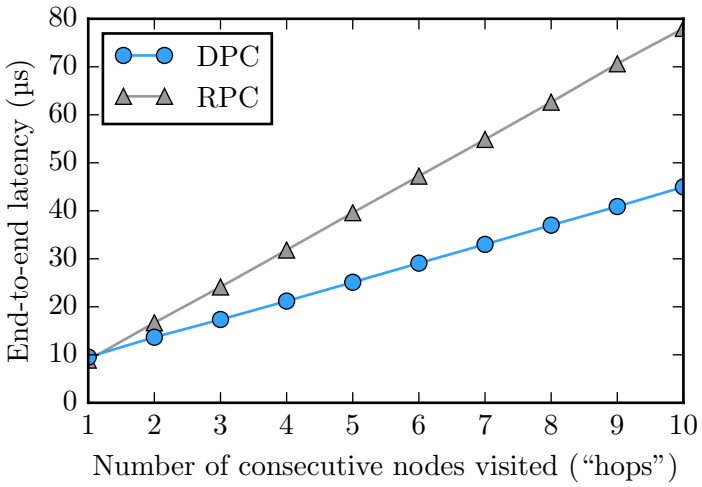


Figure 6.1: The median end-to-end latency as a function of the number of consecutive nodes that the execution must visit. In the case of RPC, an n-hop execution consists of n serial RPCs issued by the client with 1 request and 1 response for each RPC. In the case of DPC, an n-hop execution consists of a DPC with 1 initial request from the client, $n-1$ delegated requests between servers, and 1 response back to the client. Each message in the execution contains a 100B payload.

two hops, DPC reduces latency by 18% compared to RPC and reaches a 42% reduction at 10 hops; the latency reduction asymptotically approaches 49%. The latency improvements are a direct result of the reduced number of network hops that a DPC execution must perform on the critical path of the execution. For each additional node that must be visited, the RPC implementation adds an additional RPC, consisting of two additional messages, to the critical path. In comparison, the DPC implementation only adds a single delegated request. However, the benefits of DPC are lost when used with only a single hop; with only a single request and single response, Roo's more complex protocol logic increases latency by roughly 600ns (from RPC's 8.89µs to Roo's 9.53µs).

The second parameter to consider is the degree of "fan-out." Figure 6.2 shows the latency of a 2-hop execution where the second hop requires visiting multiple nodes in parallel. This execution pattern can be found in indexed queries for a distributed object store where the first hop fetches the index information and the second hop returns some number of indexed objects from different servers.

There is not a significant difference in latency between the DPC and RPC executions because most of the time is spent sending concurrent messages to many nodes. In the case

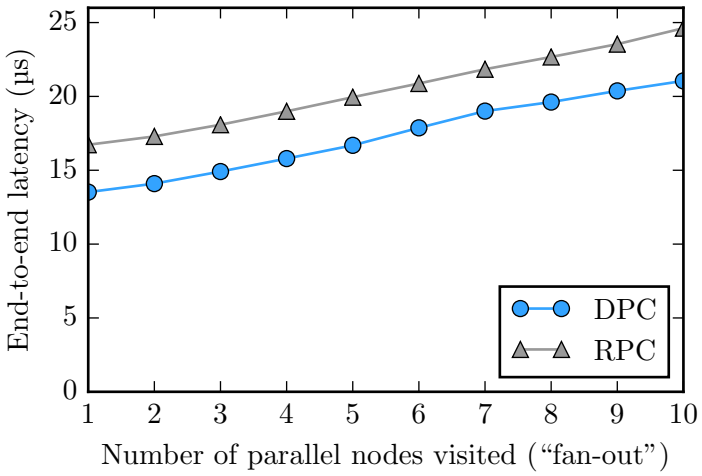


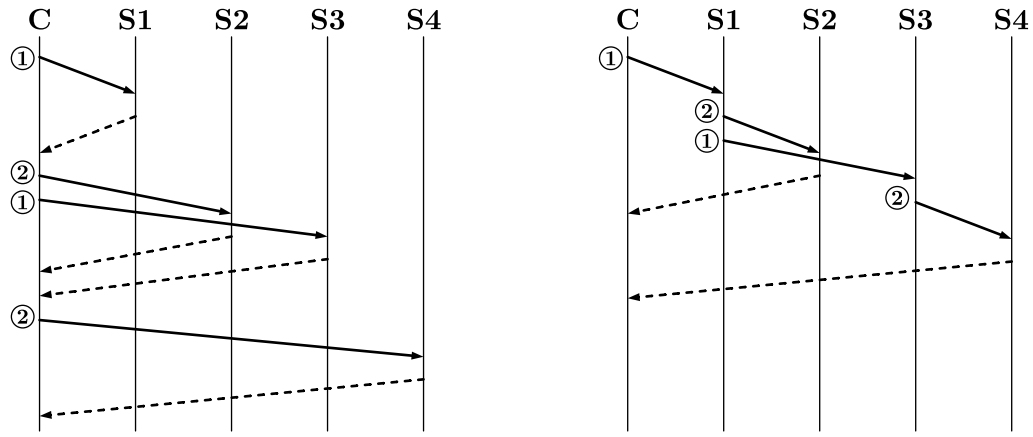
Figure 6.2: The median end-to-end latency of a 2-hop execution as a function of the number of nodes that must be visited in parallel for the second hop. In the case of RPC, an n-fan-out execution consists of a single RPC followed by n RPCs in parallel all issued by the client. In the case of DPC, an n-fan-out execution consists of a single request sent by the client to a level-1 server, n delegated requests sent by the level-1 server to n level-2 servers, and n responses sent back to the client (1 for each of the n level-2 servers). Each message in the execution contains a 100B payload.

of RPC, this is done by the client. In the cases of DPC, this is done by the server. The only difference with DPC is that the concurrent messages start one network hop sooner since an intermediate message is eliminated. This roughly 3µs latency reduction, essentially the savings from a 2-hop execution, is consistent for all degrees of fan-out. Otherwise, total latency increases at the same linear rate for both DPC and RPC as the cost of each additional concurrent message is the same. Because the absolute benefit remains the same regardless of fan-out, the relative benefit of DPC degrades as higher degrees of fan-out increase the total latency.

Examining the effect of both the number of hops and the degree of fan-out suggests that DPC sees the best performance improvement with larger numbers of hops and small to moderate levels of fan-out.

6.3 Distributed Ray Tracing

With a potentially large number of hops and a small degree of fan-out, distributed ray tracing is a good candidate application for DPC. Ray tracing [37] is a technique for rendering



(a) RPC **(b)** DPC

Figure 6.3: The communication pattern for a two-bounce execution of distributed ray tracing implemented with (a) RPC and (b) DPC. Intersection calculation and shadow ray processing requests are labeled (1) and (2) respectively.

3D images by simulating light as it bounces through objects in the scene. A popular approach is to start at the camera and trace the path of light rays backwards. In distributed ray tracing [7, 16] scene data and execution is split between a large number of machines. After an initial set of rays is generated starting at the camera, the rays must be processed. Processing a single *ray* involves first determining if the ray intersects or "hits" an object and then determining if a hit object is illuminated by checking if a *shadow ray* can reach a light source. A ray hit also generates a new ray that simulates light bouncing. The process is repeated for the new bounced ray and continues until a ray fails to hit an object or some configured bounce limit is reached. Since the scene data is split across many machines, simulating a single ray involves serially communicating with a number of machines as each bounce may need scene data from a different node.

I evaluated both DPC and RPC using a benchmark that models the communication needed to simulate a path of a single ray. Calculating each bounce of the ray involves two sequential requests. The first request determines if a ray hits an object (i.e. object intersection calculation). If a hit is confirmed, a second request determines whether the hit object is illuminated (i.e. shadow ray processing). Thus two hops are needed when only simulating a single bounce. With each additional bounce, two additional requests are needed. However, the processing of the additional bounce can be partially run in parallel

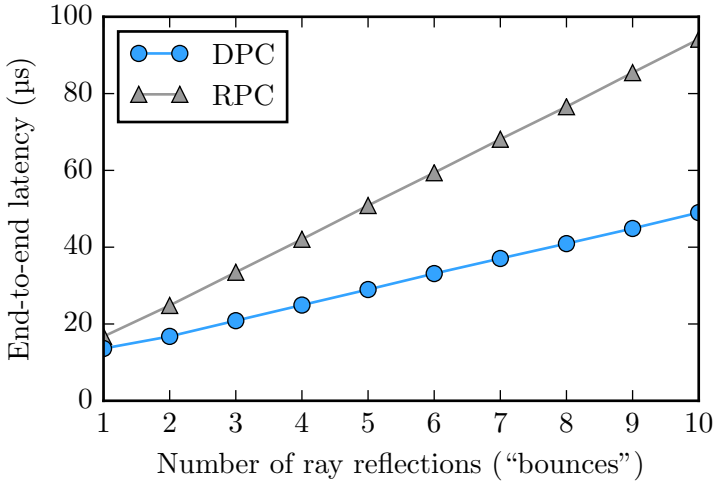


Figure 6.4: The median end-to-end latency of a communication pattern modeled after a ray tracing execution as function of the number of ray "bounces" that need to be traced. Each message contains a 100B payload.

with the processing of the previous bounce. This is because a bounce can begin executing as soon as the previous bounce has completed its intersection calculation. As a result, the intersection calculation of a new bounce can be processed in parallel with the previous bounce's shadow ray. This means that a single ray starts with two hops with each additional ray bounce adding one request in parallel (i.e. a fan-out of two) and one additional hop. In practice, each object intersection calculation and shadow ray processing step itself may involve multiple hops since the scene data is represented in a tree structure [32] but the model is simplified to assume only a single hop for clarity. Figure 6.3 shows the resulting communication pattern for both RPC and DPC for a two-bounce execution.

Figure 6.4 shows the execution latency for a varying number of ray bounces. The performance is dominated by the number of hops involved in the execution so the results are similar to the "hops" experiment in Figure 6.1. However, the latency reductions are actually slightly better than in the simple "hops" experiment; the latency in this experiment is reduced by 48% at 10 bounces and the reduction asymptotically approaches 54%. The reason is the RPC client needs to process far more messages than the DPC client. With n bounces, the RPC client needs to process $4n$ messages (2 RPCs per bounce) compared to the $n+1$ messages processed by the DPC client. This makes the RPC client a performance bottleneck. This kind of client bottleneck is explored in more detail in Section 6.5. In a real execution of distributed ray tracing, the number of bounces would fall in the range of

Query	Description	Hops	Fan Out
IS 1	Given a start Person, retrieve their first name, last name, birthday, IP address, browser, and city of residence.	1	2
IS 2	Given a start Person, retrieve the last 10 Messages created by that user. For each Message, return that Message, the original Post in its conversation, and the author of that Post. If any of the Messages is a Post, then the original Post will be the same Message, i.e. that Message will appear twice in that result.	4	20
IS 3	Given a start Person, retrieve all of their friends, and the date at which they became friends. (Assume 100 friends with data split across 10 servers.)	2	10
IS 4	Given a Message, retrieve its content and creation date.	1	1
IS 5	Given a Message, retrieve its author.	2	1
IS 6A	Given a Post, retrieve the Forum that contains it and the Person that moderates that Forum.	3	2
IS 6B	Given a Comment, return the Forum containing the original Post in the thread that the Comment is replying to and the Person that moderates that Forum. (Assume Comment is direct reply to Post, i.e. 1-hop away.)	4	2
IS 7	Given a Message, retrieve the (1-hop) Comments that reply to it. In addition, return a boolean flag indicating if the author of the reply knows the author of the original message. If author is same as original author, return false. (Assume 10 Comments and 100 friends via the knows relationship with data split across 10 servers.)	3	21

Table 6.2: The description for each of the LDBC SNB Interactive Short Read Queries [8]

3-10 but since each processing step might involve several hops, the total number of hops in practice would be several times higher. As a result, the latency reductions would rapidly approach the 54% asymptote in practice.

6.4 Graph Queries

DPC is also well suited for distributed graph databases and their queries. Graph database queries are modeled as an execution that traverses a graph, computing over node and edge data as it goes. In a distributed graph database, the node and edge data needed for any particular query is often spread across multiple machines (partitioning graph data to maximize

data locally is well known to be a hard problem). As a result, queries involve communicating with a number of nodes both in series and in parallel as the graph is traversed.

To evaluate both DPC and RPC applied to graph queries, each is used to implement the execution pattern of queries drawn from the LDBC Social Network Benchmark [8], an industry-standard graph database benchmark. The benchmark models a graph representation of a social network and defines common queries executed by users and operators of the social network. The LDBC's suite of "Interactive Short" (IS) queries was selected for this benchmark; the query suite represents common interactions a user might have with a social network application. For RPC, each execution consists of a collection of RPCs all driven by the client; this client-driven style of execution is common for many graph query implementations. For DPC, each execution consists of a single DPC and use delegated requests wherever reasonable; this eliminates the need for many intermediate response messages. Table 6.2 provides a description of each of the benchmarked queries.

Figure 6.5 shows the latency for each of the queries in Table 6.2. The number of hops involved in the query is for the most part predictive of the relative performance between DPC and RPC. In IS 1 and IS 4 where there is only one hop, RPC performs better because of its lower protocol overhead. With IS 5 and IS 6, the number of hops is mostly predictive of DPC's latency reductions compared to RPC.

The benchmark, however, shows three cases where the latency reduction exceeds what is predicted by accounting for hops alone: with two hops, IS 3 experiences a 24% latency reduction; at 3 hops, IS 7 enjoys a 44% latency reduction; and with just 4 hops, IS 2 sees a 49% latency reduction. As was the case with distributed ray tracing, this is the result of the DPC client having to process far fewer messages than the RPC client, effectively reducing a message processing bottleneck at the client. For example, executing IS 2 with RPCs involves 41 RPCs with all 82 messages processed by the client. In contrast, executing IS 2 using DPC only requires processing 61 total messages with just 21 of those (1 request and 20 responses) involving the client. By structuring the execution as a DPC, not only is the total number of messages reduced but work is naturally shifted from a single client to multiple servers executing in parallel. This set of benchmarks shows that DPC can reduce end-to-end latency in two ways: first, by reducing the number of network hops, and second, by reducing the number of messages that need to be processed (particularly the number processed by the client).

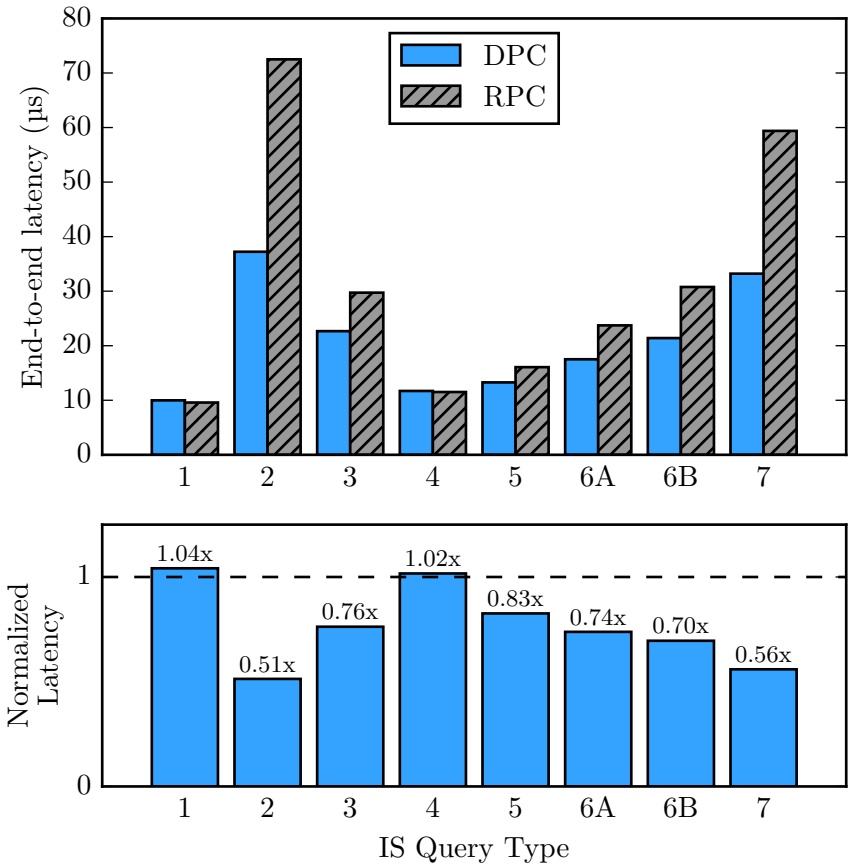


Figure 6.5: The median end-to-end latency for each of the LDBC IS queries. (Top) The absolute latency of both DPC and RPC for the various workloads; lower is better. (Bottom) The latency of DPC normalized to the latency of RPC for the various workloads; lower is better.

	IS 1	IS 2	IS 3	IS 4	IS 5	IS 6A	IS 6B	IS 7
DPC	4 (4)	61 (21)	22 (12)	2 (2)	3 (2)	6 (3)	7 (3)	54 (23)
RPC	4	82	22	2	4	8	10	66

Table 6.3: The total number of messages required to execute each of the LDBC IS queries using either DPC or RPC. In parentheses is the number of messages that the client either sends or receives; there is no number shown for RPC since all messages are either sent or received by the client.

6.5 Throughput and Resource Utilization

Although DPC is primarily focused on reducing end-to-end latency, I also evaluated DPC in terms of throughput and resource utilization. Using the LDBC IS suite of queries, DPC is analyzed in terms of the CPU utilization, the required number of messages, the protocol's byte overhead, and the achievable latency and throughput under load.

First, the CPU time required to execute each query pattern is shown in Figure 6.6. Overall, DPC and RPC require similar total CPU execution time though DPC uses less in the more complex queries that are better able to take advantage of the multi-hop pattern. Examining the breakdown between protocol processing time (each bar's lowest segment) and API execution time (each bar's middle segment), shows DPC's more complex protocol is slightly more expensive to execute, but since most queries send fewer messages using DPC there is less time spent executing API calls. The figure also shows how the CPU time is split between the client and the servers. DPC generally shifts work from the client to the server; this effect is most pronounced in IS 2 and IS 7 where a multi-hop execution pattern allows most of the communication to happen between servers. This shift allows work that would have been executed on a single client to be distributed among multiple servers. This can relieve potential bottlenecks at the RPC client. In fact, this bottleneck is what caused the increased latency in queries IS 2 and IS 7 mentioned in Section 6.4.

Another way to compare the processing overhead is to look at the number of messages involved in executing a query. This can be particularly important when the messages involved are small since the fixed cost of sending and receiving each message will become a bottleneck. This is true for the LDBC IS queries as the vast majority of messages are less than a few hundred bytes long; only IS 3 and IS 4 have messages longer than 2000B. Table 6.3 shows that DPC requires fewer messages than RPC for most queries; in the worst case, it requires the same number of messages as RPC. The most complex queries see the

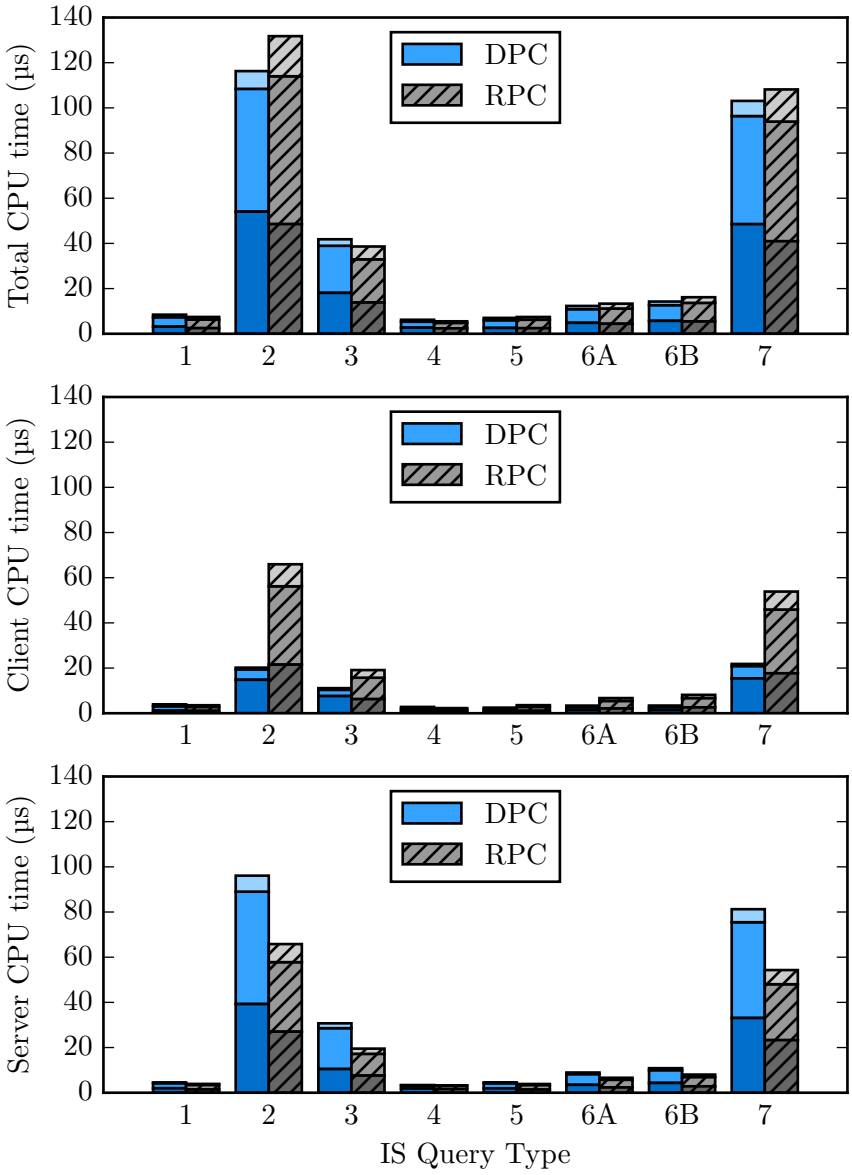


Figure 6.6: CPU usage for each of the LDBC IS queries. The three plots show the average CPU time spent in total across all machines executing the query as well as a breakdown with just the CPU time spent on the client machine, and just the CPU time spent across the server machines. Each bar is further subdivided into 3 segments: at the bottom is time spent in the transport processing the DPC or RPC protocol in the background, next is the time spent executing transport API calls (e.g. `send()`, `receive()`, `reply()`, etc.), and finally the top shows time spent in the benchmark code implementing the query execution pattern. Note that in some cases, the CPU time is so low the segment is imperceptible.

greatest reductions in message count. When considering just the messages processed by the client, the reduction is even more significant. For example, a client executing IS 2 using DPC processes about 75% fewer messages than RPC. Depending on the exact execution pattern, DPC can completely eliminate the need for some messages and in other cases can shift some of the work of message processing from a single client to a larger number of servers. This again shows the RPC client bottleneck that is relieved when using DPC.

I also analyzed the byte overhead of DPC. In Roo, each DPC message carries roughly 100B more metadata than the equivalent RPC message. This fixed, per-message overhead includes the various pieces of metadata described in Section 5 and fits in less than 7% of a standard Ethernet packet's maximum payload. When considering DPCs with small messages the performance is dominated by the per-packet processing overhead so making the payload of a single packet slightly longer has a negligible impact on throughput. With larger messages, the overhead represents a vanishingly small fraction of the bytes transmitted. While the overhead's effect on throughput is already minimal, it's likely that the size of the metadata could be cut down further. Roo allocates space for metadata fairly liberally, for example using at least a pair of 64-bit integers for all identifiers. In practice, the overhead could be cut down when considering real-world limits like the maximum DPC depth and fan-out.

Finally, I evaluated the overall throughput and latency under load. I measured the end-to-end latency of both IS 2 and IS 7 at various loads; the offered load was increased until the system reached overload. IS 2 and IS 7 were chosen since they are the two most complex LDBC IS queries and thus most likely candidates to adopt DPC. Unlike in previous benchmarks, this test was run on a cluster of 20 machines where each machine ran as both a client and a server. This prevents any bias that may result from segmenting client and server resources. The benchmark was also limited to a single core per machine so that measurements could focus on the CPU bottleneck, the primary bottleneck for queries with many small messages. Figure 6.7 shows the results of the experiment. DPC's maximum throughput is slightly better than that of RPC, which is consistent with DPC's lower CPU execution time for these queries. The latencies of both RPC and DPC increase gradually with load as the many in-flight queries share resources, causing all queries to slow down until the system reaches overload. Even so, DPC is able to maintain its latency benefit well into high loads.

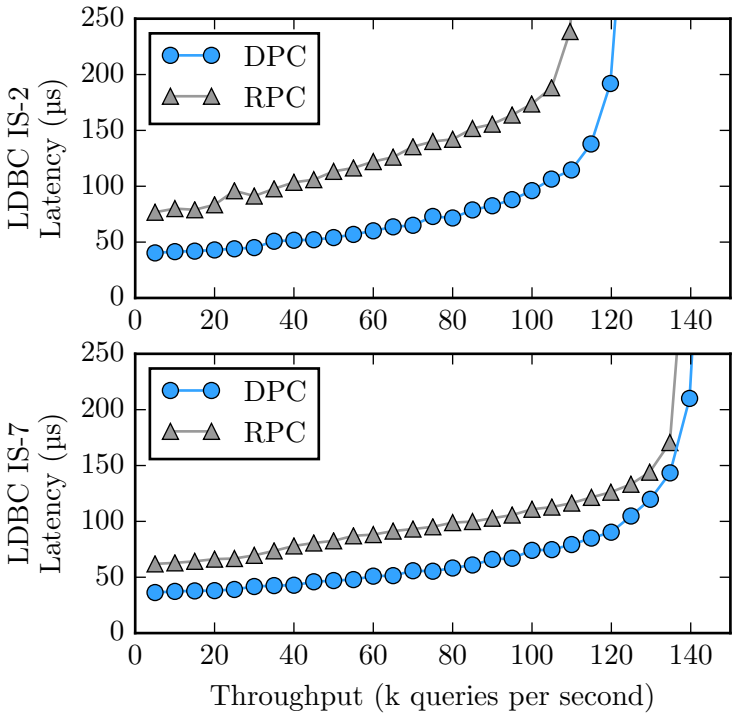


Figure 6.7: The latency of two LDBC queries (IS 2 and IS 7) as a function of total throughput of completed queries. The offered load was generated using a Poisson process.

6.6 Summary

The benchmarks show that DPC's performance improvement over RPC scales primarily with the number of consecutive nodes that need to be visited but also benefits from DPC's ability to reduce the processing load on the client. For executions involving multiple hops, Roo shows latency reductions between 18% and 49% without harming throughput.

Chapter 7

Discussion

This chapter covers a number of additional topics on how DPC might fit in different environments and how DPC could be adapted to support additional features and use cases.

7.1 Practical Implications of Coupling

DPC has a wide variety of applications, but like all tools, it's not suitable for every use case. Because a DPC is logically a single high-level operation, there is a certain degree of coupling between the various parts of an execution. This limits DPC's practical applicability to those applications that can accept DPC's level of coupling.

There are two main forms of coupling that DPC requires. First, from a software standpoint, the various program segments processing each part of a DPC are likely aware of each other or at least know that they are part of a larger DPC. For example, for one server to delegate a request to another it must at least know the other server's name and request format. Second, from a trust, authentication, and networking perspective, all the clients and servers involved need to be able to send and receive messages between each other. This means that all the server and client nodes involved in a DPC execution must be within the same network and trust domain.

Because of this coupling, DPC is probably most suitable for communication within a larger monolithic service, like a distributed database, and not as suitable for communication between a collection of decoupled micro-services. This is because of the generally accepted software engineering practice of defining a service as an independent unit of development and deployment. For example, to maintain a modular design, services may want to avoid

invoking each other directly, while DPC would require services to understand each other's APIs. The desire for decoupling between services extends to how trust domains are established and how networks are set up to limit communication across domains. For example, while an end-client, like a mobile device, might be allowed to communicate with a service gateway, most deployments would not allow arbitrary servers to communicate back out to the end-client. For this reason, a DPC client is most likely not an end-client, but rather a server within a service, which executes the DPC on the end-clients behalf. Ultimately, while DPC may not currently be a practical end-to-end solution for all applications of multi-hop communication, it is still perfectly suitable for communication within a distributed service.

7.2 WAN and High-overhead Environments

The evaluation of DPC focused on low-latency environments; this decreased the various communication costs (i.e. network software overhead and network propagation delay) so that the DPC protocol overheads would be more readily apparent. While my benchmarks have shown that DPC has benefits in these low latency environments, I believe that DPC is potentially more beneficial in environments with higher overheads.

The cost of sending messages is by definition higher in high-overhead environments. At the same time, using DPC can eliminate messages from an execution thus completely avoiding the higher costs for eliminated messages. There are two main types of cost. The first comes from software overheads like application-level processing, frameworks performing serialization and authentication, and network protocol processing, for example. In most environments today, the use of kernel-TCP and fully featured communication frameworks result in significantly higher per message overheads compared to what was represented in the benchmarks. The second cost comes from higher network latency. Network latency is especially high for geo-distributed applications where latencies can range from 10 to 100 milliseconds. However, both costs are completely eliminated for messages that never need to be sent. Thus, the higher the per-message overhead the larger the absolute benefit of using DPC.

In addition to the increased savings DPC provides in high-overhead environments, DPC's protocol processing overheads may also be lower relative to the overall execution. In high-overhead environments, the end-to-end execution time of a distributed application will likely

be higher than what was benchmarked. For example, executions might complete in milliseconds rather than microseconds. In these situations, the DPC protocol processing overhead would become an increasingly small fraction of the overall latency. As the performance cost of using DPC shrinks relative to the message processing overheads and overall latency, the benefits of using DPC should also increase.

7.3 Reliability and Retry in Unreliable Environments

To provide the illusion of reliability in unreliable environments, systems must be able to detect failures and take remedial actions like retrying the execution. DPC supports a simple abstraction where failure detection and retry happen at the level of a DPC. A DPC is considered a failure if any part of it fails and applications are expected to retry the entire DPC in response. Treating the DPC as a monolithic unit of re-execution simplifies failure handling as the retry can be triggered in one place at the client. In contrast, with an alternative design where request failures within the DPC must be individually handled, the application would need to implement failure handling at every point where a request is sent.

There are, however, two drawbacks to this simple approach that makes DPC's design better suited for typical data center environments, where failures are rare, and less ideal for generally unreliable environments. First, re-executing the entire DPC is more expensive than only re-executing the individual requests that may have failed. If failures are rare, this cost is negligible on average. However, in environments where failures are more common, the overhead can be significant. Second, treating any request failure as a failure of the entire DPC magnifies the failure rate of the DPC as a whole. A simple model of a DPC's success probability would depend on the success probability of each request and the number of requests (n). Assuming all requests have an equal probability of success, the probability of a DPC will succeed is given by the equation $\mathbf{P}(\text{DPC}) = (\mathbf{P}(\text{Req}))^n$. If a request fails only once in a million times (99.9999% success rate), a DPC with 20 requests would succeed more than 99.998% of the time and a DPC with 40 requests would still succeed more than 99.996% of the time. However, if requests fail as often as once every thousand times, DPCs with 20 to 40 requests would see failure rates in the 2%-4% range. Because of these drawbacks, DPC's design is best suited for the data center environment where network packets are rarely lost, individual servers are expected to run for days if not months without failing, and services are designed to quickly recover if failures do occur.

To be efficient in more unreliable environments, DPC would need to used additional techniques that support more fine-grain failure detection and retry. If the unreliability comes in the form of network packet drops, DPC implementations can rely on reliable network transports, like TCP, for message delivery; this would not require any changes to the DPC protocol itself. If the DPC implementation can assume DPC executions are deterministic, the protocol can be changed to allow retries at the request level and simply filter out potential duplicate responses at the client; this change would not affect the DPC API. In the most general case, with arbitrary request-level re-execution, the protocol would need a way to handle the existence of different versions of a DPC execution tree since a re-execution can generate different requests and responses. This would affect the completion detection mechanisms but might also require special handling of responses to avoid providing the client with an inconsistent or duplicated set of results. As it stands, for unreliable environments with a need for arbitrary re-execution, using a collection of RPCs may be simpler since each RPC is an independent re-executable unit, giving applications more granular control when individual requests fail. However, in data center environments or situations where high reliability can be provided using other techniques, the costs of failure are small compared to DPC's normal case benefits.

7.4 Alternatives to Manifests

The DPC protocol I described in Section 4 is not the only design that can support the DPC abstraction. In particular, there are alternative mechanisms that could be used in place of manifests. While the DPC protocol uses manifests to allow a client to incrementally discover a DPC's dynamic execution topology, the alternatives I will describe use various techniques to avoid the need to know a DPC's topology. Each of these alternatives has its own benefits and drawbacks compared to the manifest mechanism. While I ultimately decided the manifest mechanism was the most generally applicable, other implementations of DPC may find alternative mechanisms that are better suited for their particular use cases and environments.

One way to eliminate the need for manifests is to ensure the client knows enough about a DPC's execution to deduce completion without collecting topology information. To do this, DPCs must be more constrained to make them predictable. Broadly speaking, DPCs can be constrained in two ways to make them predictable for clients. The first is to make the

DPC execution pattern (i.e. the shape of the request tree) fixed, deterministic, or otherwise known at the time it is invoked. If this is the case, the set of expected responses can be pre-calculated with response identifiers pre-assigned. The second is to make the output structure of a DPC predictable. For example, if the output of a DPC execution is known to be a structure with a known number of fields, the DPC can be considered complete once all the fields of the structure are filled. In both cases, no manifests are needed since the client knows enough to form its own completion detection criteria. While there may be applications that have such properties, I believe the requirements are too restrictive for all applications in general. In contrast, the manifest-based design supports arbitrarily and dynamically generated DPC execution patterns.

Another alternative involves using a weighted acknowledgment mechanism for completion detection. The basic idea is that a DPC expects to receive one unit of acknowledgment, but each response only carries a fraction of the acknowledgment. Assuming that responses are assigned fractional weights such that the sum of all weights is one, DPC completion can be declared when the client has received a full unit of acknowledgment. Such a weight assignment is created by starting with one unit of acknowledgment and dividing it among outbound requests and responses. Initially, the client will divide its full unit of acknowledgment among the outbound requests. When a server processes a request, it will divide and assign its fraction of acknowledgment to each of its delegated requests and responses. This ultimately distributes the initial unit of acknowledgment among the response messages. Instead of a manifest, messages now must carry a representation of a fractional number. Instead of tracking manifests and received responses, this mechanism detects completion by adding weights. However, these weights must be exact and cannot be treated as normal floating-point numbers. This is because any form of rounding will result in incorrect completion detection. This presents challenges both in terms of how these weights should be represented and how addition is performed; an implementation might require variable-length weight representations and will require a way to perform lossless addition. Assuming these challenges are overcome, the benefit of this mechanism is that it is logically simpler than the manifest-based mechanism, however, it will have to be implemented to determine whether it will actually perform better in practice.

One benefit of the manifest mechanisms not present in these alternatives is that the collected topology information can be used to reduce the amount of pinging that needs to occur when failure detection is triggered on slow DPCs. Manifests allow the client to find

out which requests have already been completed and learn the identity of the intermediate servers for incomplete requests. Without manifests, the alternative mechanisms would force failure detection to walk the entire execution tree as there is no way to know that an individual request is complete and what server needs to be pinged to check its status. By leveraging the incrementally gathered topology information, failure detection can be performed with much lower overheads. This can make it more practical to trigger failure detection more aggressively. For example, instead of setting a ping period that must be long enough for 99.99% of executions to avoid the cost of failure detection, it can be set long enough for just 99%. This could greatly increase how quickly real failures are detected.

There are potentially other uses for the topology information provided by manifests. Like, failure detection, the abort protocol is also more efficient with manifests. Manifests might also be useful for debugging and tracing if the topology information were exposed through some API.

While the manifest mechanism is not the only way to support the DPC abstraction, it is more suitable than the alternatives for general data center applications. If DPC execution is predictable, clients can use that knowledge instead of manifests to detect completion. If it is exceedingly rare for DPCs to fail or be slow, perhaps the weighted acknowledgment mechanisms could be simpler. However, for general data center applications where DPC executions can be dynamically defined and stragglers are not uncommon, I believe the manifests mechanism strikes the right balance between complexity, flexibility, and efficiency.

7.5 Expanding on DPC

The focus of this work has been developing the core protocol necessary to support DPC's multi-hop communication pattern. This left a number of interesting topics unexplored. Here I briefly describe some potentially interesting ways in which to expand on DPC.

7.5.1 Programming Patterns and Interface Definition Language

While I have shown a few example applications, it's hard to know what common programming patterns will emerge for DPC. Still, based on my experience thus far, it seems less likely that DPC execution patterns will be statically defined and more likely that DPCs will be dynamically or programmatically generated.

There are a couple of dynamic patterns that I have already observed. The first is where

the application logic is recursive in nature, where processing a request may result in the delegation of the same request with different parameters. Both Chord and Distributed Ray Tracing have recursive application logic. The second is where the phases of execution are fixed but the parameters dynamically affect the selection of target servers. This is the case for the index lookup and graph query use cases.

More complex topologies may emerge if DPC logic is not created directly but rather generated programmatically. Using a compiler of sorts would allow executions to be defined at a higher level and then translated down into more complex executions. For example, perhaps a database query is defined in a declarative query language where a query engine can translate the query into a DPC execution. This could allow for more complex DPC executions than developers might want to build by hand.

Depending on what patterns emerge, a compatible interface definition language (IDL) can be developed. Today, IDLs exist for the RPC abstraction. However, given the difference between RPC and DPC, an IDL for DPC would likely be different as well.

7.5.2 Scheduling

Job scheduling policies can be used to affect various aspects of an application's performance. For example, when a system must process a number of incoming jobs, scheduling jobs using a run-to-completion policy can improve the average end-to-end latency of jobs compared to a fair sharing model. Whereas a fair sharing schedule would interleave job execution and cause all jobs to have a delayed completion time, a run-to-completion schedule would allow each job to complete as quickly as possible with each job delayed by the jobs that ran before. While this can result in unequal execution delays, it can improve the latency of most jobs since most jobs would experience less delay. Different scheduling policies could be adopted depending on an application's goals.

In non-batch-oriented distributed executions like those supported by DPC, the unit of scheduling is typically the request. A server might have a collection of incoming requests and a scheduling policy might dictate when each request will be executed. The simplest model, which Roo adopts, is FIFO processing of requests, which allows each request to run to completion.

Unfortunately, a request-level scheduling policy does not necessarily translate to the equivalent policy at the level of the entire distributed execution. For example, request-level FIFO processing results in something closer to fair sharing at the DPC level. This is

because a DPC execution is composed of many requests where some requests are generated only after processing others. If there are multiple concurrent DPC executions, a FIFO request execution order could still result in interleaving between DPC executions.

To control the execution behavior at the level of entire distributed executions, scheduling policies must be defined on DPCs rather than individual requests. A DPC-level policy could then be translated back down to request-level policies. For example, to approximate FIFO at the DPC level, a DPC's requests could include the timestamp of when the DPC was created. Then servers could prioritize request execution based on this timestamp. This would allow older DPCs to finish executing before newer ones continue. This is just one possible policy where DPC executions can be more deliberately scheduled to achieve some end goal and it would be interesting to explore what other policies could be developed.

7.5.3 Distributed Pipe

I previously stated that DPC was not suitable for inter-service communication, for example coordinating an execution involving many independent microservices. However, DPC might be able to support a higher-level execution model that can both take advantage of the multi-hop pattern and incorporate independent services.

The model takes inspiration from executions using the Unix *pipe*. In Unix-like operating systems, larger executions can be constructed by stringing together a collection of independent programs and connecting the output of one program with the input of another. In this model, the programs have no knowledge of each other and their relationship is only defined by the pipes between them. In a similar fashion, I imagine a distributed pipe-like execution system could be developed. Using this execution layer, the responses of one service could be forwarded as requests to other services. Since messages could flow directly from service to service in this way, DPC would fit naturally as the communication layer of such an execution system.

Chapter 8

Related Work

This chapter discusses DPC through the lens of related work to better describe the various features of DPC and how DPC fits in the broader context of communication systems for distributed systems.

8.1 Communication Primitives

Communication primitives are most useful for distributed systems when they map closely to an application's style of distributed execution. Because there are many styles of execution, there are also many types of communication primitives. DPC is best suited for distributed executions that involve multiple servers and where the execution pattern is formed dynamically based on the data involved in the execution. As we will see, this contrasts with the other styles of execution that are better supported by existing communication primitives.

RPC [11] is perhaps the most common communication primitive for distributed systems today. However, unlike DPC, an RPC does not fully encapsulate a distributed execution, only a remote one. RPC is structured as a pair of messages between two machines: RPC provides bi-directional communication between a client that initiates communication and a server that responds. RPCs are best for invoking API calls where the caller communicates with and expects a response from a single callee, which is the case for most modern "service" boundaries. However, if an execution is distributed, involving multiple servers, multiple RPCs would need to be used. As I've described before, this can lead to communication patterns that require extra messages and increases end-to-end latency. DPC, on the other hand, natively supports distributed executions.

The message queue is also a common primitive for distributed executions. Message queue implementations like ZeroMQ [18] and RabbitMQ [36], provide long-lived, one-to-any, bidirectional communication channels. These channels can be used to set up direct server-to-server communication to create a number of different communication patterns, including patterns similar to the multi-hop request tree. However, because channels are long-lived, the server-to-server communication pattern is likely determined a priori. In contrast, DPC allows servers to make arbitrary decisions about the communication pattern dynamically as part of the execution. This allows the server to take into account the specific data being processed in its delegation decision. For example, in an indexed lookup, the index server could use the index information to select the target server for the delegated lookup request. Where communication patterns based on message queues are essentially static, DPC allows the communication pattern to change from execution to execution.

Another popular communication primitive is the pub-sub or publish-subscribe model of communication; there are many example implementations including Apache Kafka [23], Amazon Simple Notification Service [1], Google Cloud Pub/Sub [24], and Azure Event Hubs [5]. The pub-sub abstraction allows any number of server nodes to produce messages that can be labeled and published under designated topics. Then any number of server nodes can subscribe to the topics to receive the published messages. In a similar way to message queues, executions can form communication patterns by publishing and subscribing to topics. Like with message queues, pub-sub topics are typically long-lived. However, unlike message queues, which provide a direct link between servers, a topic is a logical relationship between publishes and subscribers. This makes it easier to support multiple communication patterns by using a different pre-configured set of topics for each pattern. However, to support arbitrary patterns chosen at run time, enough topics would need to be configured to support every possible communication pattern. In practice, this would be far too many topics to manage. Most pub-sub systems use relatively static communication patterns and support pipeline-style distributed executions. In contrast, DPC requires no pre-configuration and can naturally support arbitrary communication patterns.

DPC supports a style of communication and distributed execution that isn't well supported by any other existing primitive. It allows communication patterns to be defined dynamically where patterns can even depend on the data being processed. While DPC certainly does not replace the existing primitives in their respective use cases, it does provide a new primitive for the previously unsupported multi-hop style of execution.

8.2 Limits of the RPC Abstraction

DPC is not the first work to notice the potential inefficiencies of RPC's restrictive request-response pattern of communication. There are other systems that try to eliminate some of these inefficiencies. However, all of these systems choose to retain the RPC abstraction. This choice limits these systems, restricting their benefits to only specific use cases or requiring expensive mechanisms to retain the illusion of an RPC abstraction. In contrast, DPC's defining choice was to define a new abstraction that directly supports the multi-hop pattern. This allows DPC to handle all the cases supported by these systems while maintaining low overheads.

There are three example systems I will highlight. First is R2P2 [22], a system designed to improve the performance of RPC when intermediated by middleboxes like proxies and load balancers. In this common architecture, a client sends an RPC to a middlebox, which then makes a nested RPC call to an actual server on the client's behalf. This results in a client request completing in 4 network hops when it could be accomplished in as few as 3 hops. R2P2 eliminates the extra hop by allowing different packets within an RPC to be routed through the network differently. With R2P2, the initial packet of a client's request is routed through the middlebox and all other packets can be routed directly between the client and server. R2P2 maintains the RPC abstraction for both the client and server and allows a middlebox to make routing decisions at a level below. While this can improve performance for proxied and load-balanced RPC communication, it falls short of allowing arbitrary application code to execute on an intermediate server.

Next is Cap'n Proto [3], a general-purpose RPC implementation that tries to reduce RPC overheads in a number of ways. One such technique eliminates unnecessary messages and network hops from the execution of simple composed RPCs where the result of an RPC is used directly as an argument of another. For example when a traditional RPC system is used to invoke `foo(bar())`, the result of `bar()` would need to be returned to the client only to be repackaged in the request for `foo()`; this results in extra messages and latency. In the special case where both `foo()` and `bar()` are directed at the same server, Cap'n Proto optimizes the execution by completely eliminating the intermediate messages and passes the return value of `bar()` directly to `foo()` without leaving the server. This technique is another example of how breaking from RPCs strict communication pattern can reduce latency but only has limited applicability.

Finally, the system most closely related to DPC is RPC Chains [33]. Like DPC, RPC Chains eliminates excess messages by allowing an execution to flow directly from server to server. However, it does so without breaking away from the RPC abstraction. To achieve this, RPC Chains internally performs function shipping, bringing the execution to the server instead of returning a message back to the client. Take, for example, a nested call where `foo()` calls `bar()` in the middle of `foo()`'s execution. RPC Chains could execute `foo()` until the point where `bar()` is called. When the request for `bar()` is made, the code for `foo()` is shipped along with it, allowing the remainder of `foo()` to finish executing on the same machine that processed `bar()`. While RPC Chains is ultimately able to achieve the same communication patterns as DPC, the need to maintain the RPC abstraction forces the system to do function shipping. For the low latency execution environments that DPC tries to support, the cost of function shipping could outweigh any potential benefits of the more optimal communication patterns.

In all three systems, retaining the RPC abstraction prevents each of them from providing the full benefits of a multi-hop execution to their applications. For R2P2 and Cap'n Proto, applications would only see benefits from very specific and simple execution patterns. For RPC Chains, arbitrary execution patterns were supported but the performance is hamstrung by the need for an expensive function shipping mechanism. By adopting a new abstraction, DPC allows applications to directly express multi-hop execution patterns, allowing a greater variety of communication topologies with significantly lower overheads.

8.3 Higher-level Frameworks

While the design of DPC and the implementation of Roo focus exclusively on the protocol necessary to provide the general abstraction for multi-hop communication, Roo could be extended to also support various "framework-level" features. RPC frameworks like gRPC [6] and Thrift [2] include higher-level features like message serialization/deserialization, authentication, and application threading. Future DPC libraries could provide similar features and perhaps even application-specific solutions for various fault-tolerance techniques. These DPC frameworks would further simplify the adoption of DPC.

8.4 Raw Performance Improvements

The performance of distributed executions can depend on the cost of communication. DPC tackles these costs by fundamentally eliminating the need for some communication. However, there is a large body of work focused on trying to minimize the communication cost instead. Some systems provide general RPCs and use various techniques to optimize the performance of their implementation [20, 19, 31]. Other works focus more generally on providing higher performance network transports [9, 30, 17, 28]. These various techniques for improving communication latency are orthogonal to the benefits provided by DPC and so they can be applied to DPC implementations to further improve performance.

Chapter 9

Conclusion

This dissertation set out to show that Distributed Procedure Call can provide a simple abstraction for multi-hop communication and that taking advantage of this communication pattern can improve application performance. Along the way, I have described the DPC abstraction, its concepts, and its API in hopes of helping application developers understand how DPC can be used. I have defined the DPC protocol and provided details of how Roo, the reference DPC library, was implemented so that others might also build their own custom DPC implementations. And finally, I have measured DPC's performance against RPC, showing that DPC can reduce end-to-end latency by 18% to 49% without affecting throughput, even in an already low-latency environment.

As evidenced by the DPC protocol, managing multi-hop communication efficiently is not a simple task. Clients initiating a multi-hop execution must be able to determine which of the incoming messages contain the results of the execution and also determine if and when the execution has completed or failed. This requires metadata about the distributed execution to be collected and returned back to the client. To keep this management overhead low, DPC collects and returns this metadata incrementally and only when convenient or urgent. This allows the DPC protocol to manage multi-hop executions with no extra protocol messages in normal case execution, ensuring the most efficient communication.

While the protocol is complex, DPC implementations can hide this complexity behind the DPC abstraction. The abstraction itself is relatively simple and allows applications to directly define their desired multi-hop execution patterns using the concept of delegation. When DPC is provided as a reusable library, like Roo, applications can easily take advantage of multi-hop without ever worrying about the protocol's complexity.

By supporting multi-hop communication, DPC can provide better performance for a wide variety of applications. Even in the simplest multi-hop executions involving just two hops, DPC can decrease end-to-end latency by 18% compared to using RPC. In more complex executions, DPC shows even more significant improvements. For example, using DPC for a complex graph query pattern involving 20-way fan out with as many as 4 hops provides a 49% reduction in latency. These performance benefits apply to any application that can be structured as a multi-hop execution.

All together, DPC provides applications with a new way to easily define and build distributed executions. I have shown a number of examples of applications that can take advantage of a multi-hop style of distributed computing. However, with this new tool at their disposal, I hope developers will see new and better ways to design and build distributed systems. While it is too soon to know how useful DPC will ultimately be, I am hopeful that new use cases will emerge as developers explore new ways of designing distributed systems.

9.1 A Call to Develop New Tools

The work that would become DPC did not start off with the goal of creating a new primitive. Instead, my original intent was to explore various communication patterns that could be implemented using messaging. However, as I explored the patterns that I would come to call multi-hop communication, I began to see the mounting complexity needed to support what felt like a simple pattern of communication. It seemed impractical and improbable for developers to use communication patterns like multi-hop if it also meant taking on the complexity of implementing a protocol like what would become DPC. That is, of course, unless this complexity could be hidden behind a simple abstraction. The goal of DPC would become hiding the complexity of multi-hop communication while preserving the communication pattern's natural performance benefits. Ultimately, this would not have been possible without creating a new abstraction and a new primitive.

When new communication abstractions and primitives are introduced, a common question is "why do we need a new one?" The impetus of this question is often the sense that the primitive's features could be implemented by using an existing primitive. This sense is reinforced by our natural gravitation towards familiar tools. It's much easier to continue using the tools we have rather than investing in the development of new tools.

This same tendency to gravitate towards existing tools also leads us to cast our problems

in terms of what our tools can solve. The idea of Maslow's Hammer [29], sometimes called the Law of Instrument, tells us that if our toolbox consists of RPCs, message queues, and pub/sub, we will imagine, design, and build distributed systems in terms of these primitives even if they are ill-suited for the job. Our architectures are unconsciously bound by the limitations of our tools.

The way to break out of this limitation is to expand our toolbox, to build new tools that give us new ways to build distributed systems. DPC can be such a tool, but it should not be the only one. Whenever we design new distributed architectures, I encourage us all to take a brief moment to step back and ask ourselves, "do I have the right tool for the job?" We will not always need a new tool; when we do not, we will be glad others have built something that is useful for us. But once in a while, you will find the right tool for the job doesn't yet exist. When you do, I hope you will make your contribution and help give us all a new way to think about distributed systems.

9.2 Toward General Distributed Computing

Broadly speaking, distributed computing falls into two groupings. First are the applications that harness the collective power of many machines in a highly coordinated fashion using tightly controlled programming models; applications using Map-Reduce [13] and Spark [38] are classic examples. In these cases, developers are building complex distributed applications but have little control over the details of the execution. Second are those applications that are structured as a collection of loosely coupled nodes; the microservice architecture is an example of this in the extreme. Here developers have complete control of the execution but are typically focused on the execution within a node rather than coordinating the distributed execution as a whole.

Both types of applications enjoy a wide range of support and tooling. There is, however, limited support for a third style of distributed computing where developers can build complex distributed applications with complete control over the distributed execution. This style of general distributed computing may give developers the flexibility to explore new distributed system architectures and potentially unlock distributed computing for previously neglected applications that don't fit any existing molds.

In many ways, DPC is a tool that can support general distributed computing. DPC

treats a distributed execution as a first-class concept and allows developers to control the exact pattern of execution. This not only simplifies the development of distributed executions but also provides a framework for thinking about distributed computation.

Of course, even if DPC becomes widely adopted, it is only a step toward truly general distributed computing. I am hopeful that other tools, design patterns, and general support will continue to be developed to further enable general distributed computing.

9.3 Final Comments

In this dissertation I have shown how leveraging a different communication pattern can greatly improve the performance of a distributed system and that it can be done through the use of a relatively simple, but different, abstraction. Distributed Procedure Call, with the multi-hop communication pattern it supports, reduces end-to-end latencies by 18% to 49%, even in low-latency environment. DPC makes these performance improvements accessible to developers by providing a simple and general-purpose communication abstraction that can be practically implemented as an efficient library. In the end, my hope is that DPC will give developers a different way to think about distributed system design and that DPC will become another useful tool in the distributed systems developer's toolbox.

Bibliography

[1] Amazon Simple Notification Service. https://aws.amazon.com/sns. 75

[2] Apache Thrift. https://thrift.apache.org/. 1, 77

[3] Cap'n Proto. https://capnproto.org/. 1, 76

[4] Data Plane Development Kit. https://www.dpdk.org/. 6, 37, 52

[5] Event Hubs. https://azure.microsoft.com/en-us/services/event-hubs/. 75

[6] gRPC. https://grpc.io/. 1, 77

[7] R2T2. https://github.com/r2t2-project/r2t2. 8, 56

[8] The LDBC Social Network Benchmark (version 0.3.2). http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf. 8, 58, 59

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013. 6, 78

[10] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association. 6

[11] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984. 1, 74

[12] Brent Callaghan, Brian Pawlowski, and Peter Staubach. Rfc1813: Nfs version 3 protocol specification, 1995. 18

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 81

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. 53

[15] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. 26

[16] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, 2019. 8, 56

[17] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015. 6, 78

[18] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013. 75

[19] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association. 6, 78

[20] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016. 6, 78

[21] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 57–70, 2016. 7

[22] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association. 76

[23] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011. 75

[24] SPT Krishnan and Jose L Ugia Gonzalez. Google cloud pub/sub. In *Building Your Next Big Thing with Google Cloud Platform*, pages 277–292. Springer, 2015. 75

[25] Collin Lee. Roo source code. https://github.com/PlatformLab/Roo. 37

[26] Collin Lee and Yilong Li. Homa source code. https://github.com/PlatformLab/Homa. 37

[27] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86, 2015. 16, 18

[28] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019. 6, 78

[29] Abraham H. Maslow. *The Psychology of science: a reconnaissance*. Gateway Editions, 1966. 81

[30] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John 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*, pages 221–235, 2018. 6, 37, 78

[31] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The

ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015. 78

[32] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016. 57

[33] Yee Jiun Song, Marcos Kawazoe Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC Chains: Efficient Client-Server Communication in Geodistributed Systems. In *NSDI*, volume 9, pages 277–290, 2009. 77

[34] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001. 7, 13, 14, 53

[35] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004. 3, 8, 53

[36] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012. 75

[37] Turner Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, pages 4–es. 2005. 8, 55

[38] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016. 81