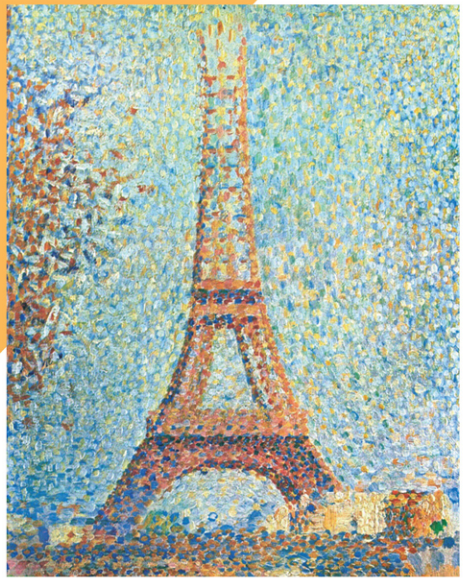
Understanding Software Dynamics

Richard L. Sites



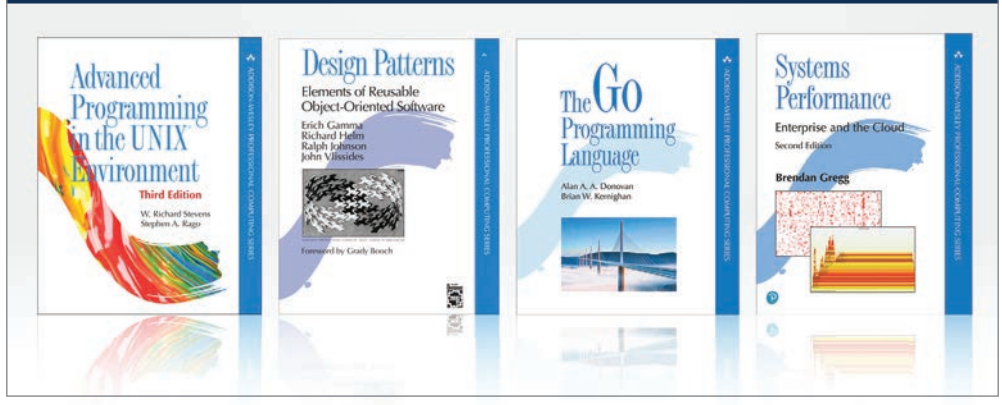
Foreword by Luiz André Barroso, Google Fellow

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Understanding Software Dynamics

The Pearson Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor



Visit **informit.com/series/professionalcomputing** for a complete list of available publications.

The **Pearson Addison-Wesley Professional Computing Series** was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. Pearson Addison-Wesley is renowned for publishing accurate and authoritative books on current and cutting-edge technology, and the titles in this series will help you understand the state of the art in programming languages, operating systems, and networks.

Make sure to connect with us!
informit.com/socialconnect

Pearson Addison-Wesley

informIT.com
the trusted technology learning source

Understanding Software Dynamics

Richard L. Sites

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021944164

Cover image: Art Heritage/Alamy Stock Photo

ISBN-13: 978-0-13-758973-9
ISBN-10: 0-13-758973-5

ScoutAutomatedPrintCode

Dedicated to the memory of Chuck Thacker,
a true Friend of the Electron who could do more
performance analysis in his head than most mortals.

This page intentionally left blank

Contents at a Glance

Contents

Foreword

Dick Sites approaches problem-solving in a way that is shockingly rare these days: he finds it almost personally offensive to make guesses, and instead he insists on understanding a phenomenon before trying to fix it. When faced with the complexity of modern computer systems, including their hardware and software, most programmers approach performance debugging armed with a hunch about what is happening and proceed to "try this, try that" with the hope that this might yield a shortcut to a solution. Those of us who use this method are implicitly giving up on the possibility of truly grasping the complex interactions that could cause a program to underperform. The idea that something computer related is beyond understanding certainly doesn't occur to Dick. Often it is the case that basic tools that provide telemetry on a program's behavior are missing. In those cases Dick does the obvious thing (for Dick), which is to build them, including the visualization framework that compresses essential information about program execution into readable charts that shine a bright light into program dynamics.

When you go through Dick's remarkable career, it becomes clear why he is confident in his ability to understand complex computing systems. He became a programmer at age 10 in 1959, and his curiosity about computing resulted in a career where he studied or worked closely with giants of our field such as Fran Allen, Fred Brooks, John Cocke, Don Knuth, and Chuck Seitz, to name just a few. His accomplishments in industry are impressively broad: from co-designing the DEC Alpha Architecture to working on Adobe's Photoshop and speeding up Google web services such as Gmail.

When I met Dick (joining DEC in 1995), he was already a legend of our field, and I had the unique pleasure of spending time with him during his Google tenure and witnessed his problem-solving approach firsthand. Readers of this book will delight in the clarity of Dick's writing and how performance debugging problems are described as mysteries to be solved through his knowledge of hardware/software interactions and sequences of clues unveiled by observing detailed traces of program execution. This is a book that will be immensely useful for programmers and computer designers alike, in no small part because there is no other book to compare it with. It is as unique as its author.

—Luiz André Barroso, Google Fellow

This page intentionally left blank

Preface

Understanding the performance of complex software is difficult. It is even more difficult when that software is time-constrained and mysteriously exceeds its constraints now and then. Software professionals have pictures in their heads of their software's *execution dynamics*: How the various pieces work and interact together over time and estimates of how long each piece takes. (Sometimes they even document those pictures.) But when time constraints are not met, we have few tools for understanding *why*—for finding the root cause(s) of delay and other performance anomalies. This is a textbook for software developers and advanced students who work on such software.

Software dynamics refers not just to the performance or execution time of a single program thread but to the interactions between threads, between unrelated programs, and between an operating system and user programs. Delays in complex software often are caused by these interactions—code blocking and waiting for other code to wake it up, runnable code waiting for the scheduler to assign it a CPU to run on, code running slowly due to shared-hardware interference from other code, code not running at all because an interrupt routine is using its CPU, code invisibly spending much of its time in operating-system services or in page-fault handling, code waiting for I/O devices or network messages from other computers, and so on.

Time-constrained software handles repeated tasks that have periodic deadlines or tasks that have an aperiodic arrival rate of new requests each with a deadline. These tasks can have *hard* deadlines for sending control signals to moving machinery (airplanes, cars, industrial robots), *soft* deadlines such as for converting speech to text on the fly, or just *aspirational* deadlines such as for customer database lookup or web-search response times. Time-constrained also applies to phone/tablet/desktop/game user-interface responses. The term *time-constrained* is broader than the term *real-time*, which often implies hard constraints.

In each case, software tasks have a stimulus or request and a result or response. The elapsed time between the stimulus and result, the *latency* or *response time*, has some deadline. Tasks that exceed their deadlines fail, sometimes in catastrophic ways and sometimes in merely frustrating ways. You will learn how to find the root causes for these failures.

The individual tasks within such software can be called *transactions*, *queries*, *control-responses*, or *game-reactions* depending on the context. Here we will use the term *transactions* to encompass all of these. Often an end-to-end task is composed of several sub-tasks, some of which run in parallel and some of which depend on the completion of other subtasks. Sub-tasks may be CPU-bound, memory-bound, disk-bound, or network-bound. They may be executing but more slowly than expected due to interference across shared hardware resources or due to power-saving strategies in modern CPU chips. They may be waiting (i.e., not executing) for software locks or for responses from other tasks or other computers or external devices. There may be unexpected delays or interference from the underlying operating system or its kernel-mode device drivers, rather than in the programmer's user-mode code.

In many situations the software involved consists of a dozen or more *layers* or subsystems, all of which may contribute to unexpected delays and all of which may be running on separate networked computers. For example, a Google web search may spread the query across 2,000 computers, each of which does a small portion of the search and then the results are passed back and prioritized. An email message arrival in the cloud may trigger subsystems for databases, network

disk storage, indexing, locking, encryption, replication, and cross-continent transmission. An automobile-driving computer may be running 50 different programs, some of which interact on every video frame coming from a half-dozen cameras, plus radar returns, changing GPS coordinates, changing 3D acceleration forces on the vehicle, and feedback about rain, visibility, tire slippage, etc. A small database system might have query optimization and disk-access subsystems using a dozen disks spread across several networked computers. A game can have subsystems for local computation, graphics processing, and networked interactions with other players.

You will learn in this book how to design in observability, logging, and timestamps for such software, how to measure CPU/memory/disk/network behavior, how to design low-overhead observation tools, and how to reason about the resulting performance data. Once you have an accurate picture of the actual elapsed-time tasks and sub-tasks for normal transactions and also for slow ones, you can see how that reality differs from the picture in your head. At that point, substantially improving the slow transactions may take only 20 minutes of software changes. But without a good picture of reality, programmers are reduced to guessing and "trying things" to reduce long delays and improve performance. This book is about not guessing, but knowing.

All of the examples, programming exercises, and supplied software in this book are written in C or C++, based on the Linux operating system running on 64-bit AMD, ARM, or Intel processors. The reader is assumed to be familiar with developing software in this environment. We assume further that the reader has some software that is time-constrained and has performance issues that the reader wants to fix. The software should already be functional and deemed debugged, with acceptable average performance—the problem is just unexplained performance variance. The reader is assumed to have an in-the-head picture of how the software runs and can on request sketch how the pieces are *supposed* to interact in a typical transaction. Finally, the reader is assumed to know a little about CPUs, virtual memory, disk and network I/O, software locks, multi-core execution, and parallel processing. Together, we will take it from there.

We explore three major themes: Measure, Observe, and Reason.

Measure. The starting place for any performance study is to measure what is happening. A numerical measurement—transactions per second, 99th percentile response time, or dropped video frame count—tells you only what is happening but not why.

Observe. To understand why some measurement is unexpectedly slow or otherwise bad but measuring the same work again is fast, it is necessary to observe in close detail where all the time is going or what processing is being done for both normal and slow instances. For the hard case of unexpectedly bad behavior that only occurs under heavy live load, it is necessary to observe over a substantial enough time interval to have a high probability of observing several slow instances and to do so in situ with minimal distortion while running full live loads.

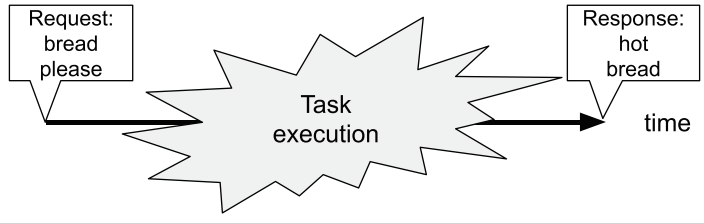
Reason (and fix). Once careful observations are available, you need to reason about what you see—how are slow instances different from normal ones, how do software and hardware interactions produce slow instances, and how can you improve the situation? In the last part of the book, we go through case-study examples of such reasoning and some of the fixes.

Following these themes, the book material is organized into four parts, including a part about building the low-overhead KUtrace observation tool:

- Part I (Chapters 1–7), **Measurement**—how to do careful measurements of the four fundamental computer resources: CPU, memory, disk/SSD, and network.

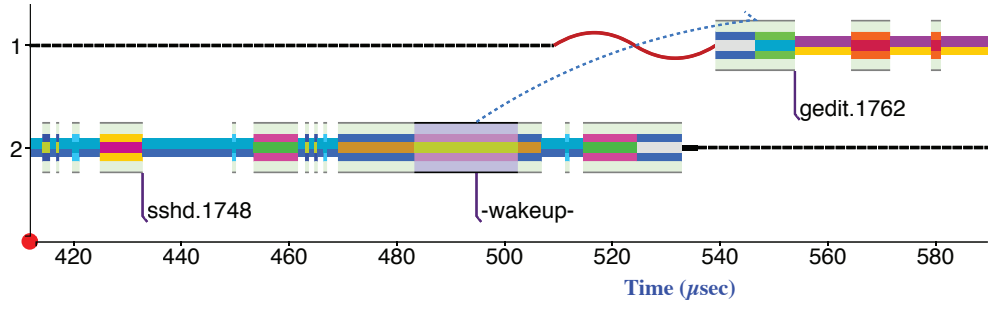
- Part II (Chapters 8–13), **Observation**—normal observation tools: logging, dashboards, counting/profiling/sampling, and tracing.
- Part III (Chapters 14–19), **Kernel-User Trace**—the design and construction of a running low-overhead Linux tracing tool that records what every CPU core is doing every nanosecond, along with postprocessing programs to create dynamic HTML pages that display the resulting timelines and interactions.
- Part IV (Chapters 20–30), **Reasoning**—case studies of reasoning about the interference underlying unusual delays observed in: excess execution, slow instruction execution, waiting for CPU, memory, disk, network, software locks, queues, and timers.

Using these ideas, you will be able to turn this picture of unexplained delay:



into the following detailed picture showing which subtasks happened when, which happened in parallel, which depended on another step finishing, and thus exactly *why* it took three hours:

The same ideas can turn an example software delay into this picture of the remote-login `ssh` daemon on CPU 2 waking up `gedit` on CPU 1:



(In Part III you will learn how to create this last kind of picture for your arbitrary software.)

This book is intended especially for engaged readers who do the included programming assignments and who implement portions of the software observation tools described.

Layered throughout this book are comments about modern complex processor chips and their performance-enhancing mechanisms. Accidently defeating these mechanisms can create surprising delays. The careful reader will gain a deeper understanding of computer architecture and microarchitecture, along with everything else.

This is a textbook for software professionals and advanced students. But it also covers material of interest to computer hardware architects, operating system developers, system-architecture IT professionals, real-time system designers, and game developers. Its focus on understanding user-facing latency will develop skills that enhance any programmer's career.

Accessing the Source Code

The book uses several computer programs: mystery1, mystery2, and so forth. The source code for these programs is available for download from Addison-Wesley at informit.com/title/9780137589739.

Register your copy of *Understanding Software Dynamics* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780137589739) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

Many people have helped along the path to this book. Amer Diwan, V. Bruce Hunt, Richard Kaufmann, and Hal Murry have actively read and provided feedback on the text. Connor Sites-Bowen, J. Craig Mudge, Jim Maurer, and Rik Farrow provided thoughtful reviews and encouragement for earlier versions and related articles. Brian Kernighan did a thorough reading of the manuscript and made suggestions to materially improve the final product.

Much of the material here was developed from graduate courses I taught after retiring from Google in 2016. I am grateful for the opportunities and the student feedback arranged by Michael Brown at the National University of Singapore; Jim Larus and Willy Zwaenepoel at Ecole Polytechnique Federale de Lausanne; Christos Kozyrakis at Stanford University; and Kevin Jeffay and Fred Brooks at the University of North Carolina.

Joshua Bakita, Drew Gallatin, and Hal Murray have done ports of KUtrace to different Unix flavors. Jim Keller and Pete Bannon provided me the opportunity to do a port at Tesla Motors. Sandhya Dwarkadas asked the key question about detecting cache interference that led to my adding instructions-per-cycle counting to KUtrace.

My early career became focused on CPU performance and tracing through the influence and guidance of Elaine Bond, Pat Goldberg, Ray Hedberg, Fran Allen, and John Cocke at IBM; Don Knuth at Stanford; and Joel Emer, Anita Borg, and Sharon Perl at Digital Equipment Corporation.

My wife of 37 years, Lucey Bowen, has been especially gracious and supportive while I spent too much time focused on completing the book.

My editor, Greg Doench, has been particularly helpful in bringing this project to a smooth completion. He took time in the early months to arrange trial runs of importing text and the extensive figures into the publishing workflow, saving time and grief near the end of the process. My copy editor, Kim Wimpsett, did a fantastic job inserting literally thousands of small improvements.

—Richard L. Sites, September 2021

This page intentionally left blank

About the Author

Richard L. Sites wrote his first computer program in 1959 and has spent most of his career at the boundary between hardware and software, with a particular interest in CPU/software performance interactions. His past work includes VAX microcode, DEC Alpha co-architect, and inventing the performance counters found in nearly all processors today. He has done low-overhead microcode and software tracing at DEC, Adobe, Google, and Tesla. Dr. Sites earned his PhD at Stanford in 1974; he holds 66 patents and is a member of the US National Academy of Engineering.

This page intentionally left blank

Chapter 6

Measuring Networks

The fourth fundamental shared resource to measure is network activity—how long do real network transmissions take and what are their dynamics? In contrast to measurements of the internal dynamics of single operations in the previous chapters, we will look at multiple overlapping network requests. The environment for disk measurements (and for that matter CPU and memory measurements) is fairly simple, as shown in Figure 6.1. There is just the one program running on one CPU and accessing one disk with a single transfer at a time.

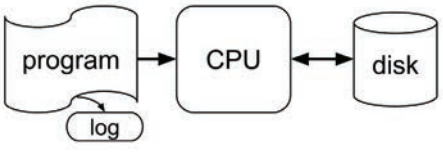


Figure 6.1 **Environment for disk measurements**

But the environment for network measurements is substantially more complicated, as shown in Figure 6.2. There are multiple client programs sending request messages to several server programs, which in turn send responses. These are all running on several different computers with network connections in between. Common server programs include database software.

In general, the different computers of Figure 6.2 could be located anywhere in the world, but we will concentrate in this chapter on computers that are physically close to each other, such as all inside a single datacenter room. The network connections could be Ethernet, Infiniband, Fibre Channel, or other choices, but we will concentrate on Ethernet connections. Various network protocols could be used, such as virtual channels, User Datagram Protocol (UDP), or Transmission Control Protocol/Internet Protocol (TCP/IP) software. We will concentrate on TCP/IP links, a common choice within datacenters.

The request messages and their responses could be structured in various ways; we will concentrate on *remote procedure call* (RPC) messages. Each RPC request message specifies a server computer to perform some work, the particular method (i.e., function or procedure name) to be called, and copies of all the method arguments. Each response message specifies the client computer to receive the response and the response data itself. The request and response messages can

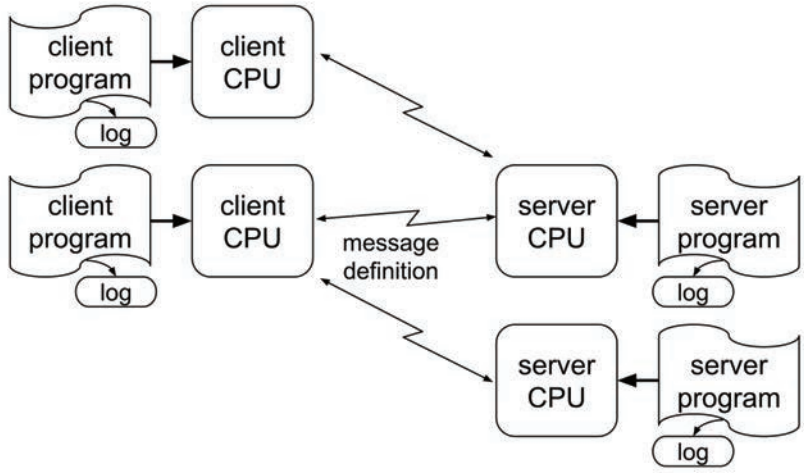


Figure 6.2 **More complicated environment for network measurements**

vary widely in size, from about 100 bytes to tens of megabytes. RPCs are usually *asynchronous*, meaning that the caller need not wait for the RPC response, but can instead continue executing and issuing other RPC requests in parallel, eventually waiting for responses that come back in arbitrary order. It is the highly parallel execution of many small pieces of work that allows datacenter software to respond quickly. Unlike TCP and other network protocols, RPC message formats are not standardized. This book uses a simple made-up format, described in Section 6.8.

In a large datacenter of 20,000 computers, with each computer running many different programs, an individual computer may have 10,000 network connections open at once, exchanging RPCs over all of them. While Figure 6.2 shows multiple point-to-point connections between client CPUs and server CPUs, these are just conceptual. The physical network may have just a single Ethernet link between each computer and a network router, with all the RPC traffic shared across these links. These underlying physical links and their associated kernel software are the shared network resource that we measure in this chapter.

> A note on notation. The word *server* is somewhat overloaded in the computer industry. It can refer to a box of hardware that is a computer, or it can refer to a program that performs some specific function on behalf of various client programs. To add to the confusion, a server program performing a specific function is often called a *service*. In this book, when the context is not clear we will refer to *server CPU* or *sample server* for a box of hardware and *server program* for a piece of software providing some service. The unqualified term *server* will generally mean a CPU.

As discussed in Chapter 1, datacenter software consists of layers and layers of subsystems, many running in parallel and often on hundreds or thousands of different servers. All this activity is tied together with some form of network message passing or RPCs. In this chapter we will observe and measure some simple RPCs, and then in the next chapter we will measure multiple

overlapping RPCs. There are several layers of software involved, including user code, the operating system, and the TCP (transmission control protocol) stack on the client computer and the same three on the server computer. We will use RPCs from one user-mode program to another and back, measuring the behavior and delays between sample servers.

6.1 About Ethernet

Ethernet is the standard networking technology worldwide and is heavily used in datacenters. The original Ethernet at Xerox PARC in 1973 used a single coaxial cable (one wire inside a tube of a second wire with insulation in between), so it was a shared medium. Individual Alto computers connected to the wire with a *vampire tap* that poked an insulated spike through the outer wire to touch the inner wire, plus a second connection to the outer wire, as shown in Figure 6.3. (The vampire taps were shown to be unreliable and were soon superseded.) Just as polite people do when talking in a group, a computer desiring to transmit would listen to the coax (carrier-sense) waiting until it was idle and then try to transmit. During transmission, it continued to listen to determine whether the bits it transmitted were on the wire or whether they were garbled because some other computer was also starting to transmit. When that happened, both would stop transmitting, each wait a random amount of time, and then try again. Any node connected to the shared coax can observe all the packets, not just those addressed to that node. This is useful for monitoring network performance and debugging network problems, but it raises security issues.

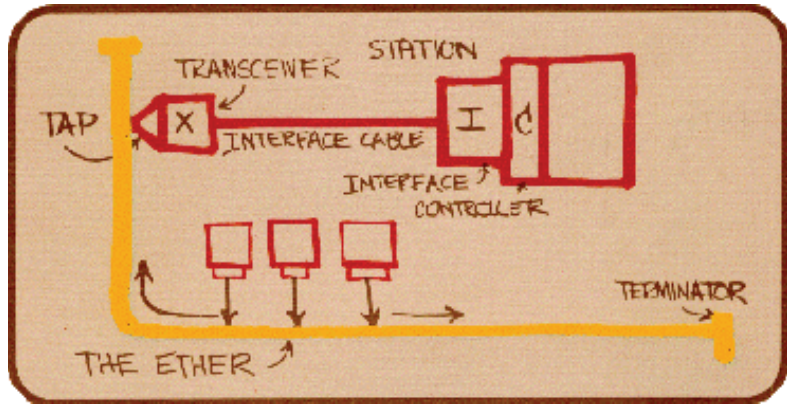


Figure 6.3 **Metcalfe's original Ethernet diagram, photographed by Boggs [Metcalfe 1976]**

Today, Ethernet data is transmitted as packets of up to 1,518 bytes (jumbo packets can be bigger) with gaps [Wikipedia 2021n] in between and a checksum at the end, as in Figure 6.4. Network software turns longer messages into sequences of packets. Individual packets are delivered with high probability, but are not 100% guaranteed to arrive. In particular, switches and routers that are overloaded are free to drop packets at any time. Packets with bad checksums are also discarded.

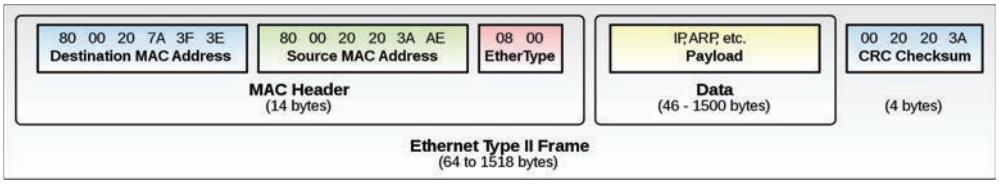


Figure 6.4 **Ethernet type II frame [Wikimedia 2020a]**

Each Ethernet packet starts with the 48-bit destination *Media Access Control* (MAC) address, followed by the 48-bit source MAC address, followed by a 16-bit *Ethertype* field and then the rest of the packet. The last 24 bits of a MAC address is an assigned Organizationally Unique Identifier (OUI) [IEEE 2021].

The remainder of a packet typically has several *headers* for layers of different switching protocols and then finally some user data. We will be using the TCP/IPv4 protocol pair, with a 20-byte IPv4 header giving the 4-byte IP addresses of the source and destination machines and a 20-byte TCP header giving the 2-byte port numbers on those machines plus data *sequence numbers* (SEQ) and *acknowledgment bits* (ACK) for accomplishing in-order guaranteed delivery.

We are using IP version 4 (IPv4) in our examples, but all the 32-bit IP numbers in this protocol are now used up worldwide, so the newer IPv6 is also being used in datacenters. IPv6 has 128-bit IP addresses, and an entire IPv6 header is 40 bytes instead of 20.

The MAC address is a unique 48-bit identifier assigned to each network interface controller in the world. (The original 3 Mb/sec Ethernet used 8-bit addresses.) The Ethertype field specifies how to interpret the following data bytes. For TCP/IP traffic, the MAC header Ethertype specifies IPv4, followed by a 20-byte IPv4 header in the first few data bytes of Figure 6.4. The IP header in turn specifies that it is followed immediately by a TCP header, which specifies that it is followed immediately by some number N of user message data or *payload* bytes.

> While the original 3 Mb/sec Ethernet used a single shared coax cable for connections, later implementations more often use twisted-pair copper wires or optical fibers running point-to-point from each computer to a hub or switch or router. These connections have progressively increased in speed from 10 Mb/sec to 100 Mb/sec, 1 Gb/sec, 10 Gb/sec, and now 100 Gb/sec with 400 Gb/sec on the horizon, *five orders of magnitude* faster than the original.

Note that network transmission rate is traditionally measured in bits per second while disk transfer rate is traditionally measured in bytes per second. Lower-case “Mb” is megabits while upper-case “MB” is megabytes. Marketing literature often confuses these, introducing factor-of-eight errors. Deliberately quoting disk transfers in Mb/sec is a cheap way to make your numbers 8x larger.

6.2 About Hubs, Switches, and Routers

Point-to-point Ethernet connections between more than two machines require some form of switching fabric. There are three different kinds that you may encounter.

A *hub* with N links is a very cheap and now rarely used design that just reproduces one incoming transmission on all inactive outgoing links. If two or more links have incoming transmissions at once, only one is copied, and the others are dropped. Since a hub can copy only one transmission at a time, it is a shared resource like the original coax.

Switches with N links store packets at each incoming port and immediately forward them to one or more outgoing ports. Smarter switches keep tables of which MAC address destinations are attached to which port and forward only to the right destination port. A switch may store as little as two packets per incoming port, forwarding one while a second arrives. If multiple packets on different incoming ports have the same outgoing port and there is not enough buffering for all of them, some of the packets are dropped. As mentioned, Ethernet does not guarantee packet delivery, just best-effort.

A *router* is a more complex form of switch, using not only the MAC address in each packet but also higher-level IP and other header address information to select the output port for each packet. Routers are often connected to other routers so that a packet may go from one end node through several routers to another end node:

$$A \Rightarrow \text{Router1} \Rightarrow \text{Router2} \Rightarrow \text{Router3} \Rightarrow B$$

Typical use in a datacenter is to have 40–50 servers mounted in a vertical rack with a router at the top (or middle) each rack. Traffic between servers within that rack is delivered directly from the top-of-rack router, while traffic destined for other racks is sent from the source top-of-rack router to one of several intermediate routers that eventually send the packet to the destination top-of-rack router and on to the destination server. Often in this case, the cross-router links run at a higher speed than the individual server links: for example, 10 Gb/sec copper-wire server links within a rack and 100 Gb/sec fiber-optic cross-rack links. We will use the phrase *on the wire* to refer to bit transmission over any kind of link. Routers often have several packets of buffering per input port, so can handle a modest amount of network congestion with several input packets destined for the same output port.

Our sample servers each have a 1 Gb/sec Ethernet port, and several are connected via a five-port switch, four ports for up to four sample servers and the fifth port connected to the rest of the building, as shown in Appendix A.

6.3 About TCP/IP

The TCP/IP design allows packets to be routed not just within a single building but anywhere in the world that is connected to the global Internet. This routing sends packets across various media—not just Ethernet links, but also long-haul dedicated fibers, radio links to satellites, WiFi connections within houses, and many more kinds of sub-networks. The complex dynamics and delays of long-distance communication are beyond the scope of this book; we will concentrate just on the complex-enough dynamics and delays of Ethernet connections within a single building.

For a message from machine A to machine B, sending software on A for *guaranteed-delivery* protocols such as TCP keeps track of packets sent that do not arrive and retransmits them. Packets are therefore not guaranteed to arrive in the order originally sent, so receiving software further tracks them and reassembles messages in receive buffers. This tracking is done by the receiving TCP software on B sending back an ACK indication to A for one or more received packets. ACKs can be sent in short packets of their own but are usually piggybacked as part of other packets already going back B⇒A. Senders have a limited number of multiple *packets outstanding*—sent but not yet acknowledged. When this limit is reached, the sender must wait until some ACKs arrive. If a packet ACK does not arrive within some configured timeout period, the sender is responsible for retransmitting that packet.

We are using TCP/IPv4 to send RPC messages between servers, with each message possibly requiring many packets. Our remote procedure calls depend on the guaranteed delivery mechanism of TCP to deliver an entire message with the pieces in proper order.

It is unlikely in our little sample server cluster that we will see packets dropped and retransmitted because of hardware errors, but we will soon try to create enough network congestion to force some packet drops because of overloaded switch buffering. To try to protect the rest of any building network from also becoming overloaded when we do saturation experiments, it is best for our lab machines to connect directly with each other through their own local switch, as described earlier.

TCP establishes a reliable connection to carry a *pair* of byte streams between two programs on two machines, one stream in each direction. These are the bi-directional connections shown in Figure 6.2. Each machine is specified by its IP address and the specific program by its port number. Two-byte port numbers range from 0..65535, but ports below 1024 are restricted to specific uses. We will use ports 12345..12348 on our sample servers for RPC traffic. (Our lab machines may also have a software firewall that closes traffic on most other ports.)

Once a connection is established, there is a stream of data available in each direction between the two machines. A machine can send an (almost) arbitrary number of bytes at once, and the TCP software deals with breaking up long messages into multiple packets, or packing multiple short messages or fragments of longer messages into single packets. The communication model is just a stream of bytes, so a given RPC message may start and end in the middle of packets.

On the other end, a machine can request receiving an (almost) arbitrary number of bytes into a buffer, but the number actually delivered at once can be less than the buffer size. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested. This design allows the receiving software some flexibility in managing buffers and in managing how long to wait for data (or what else to do in the meantime). The receiving logic thus must be prepared to do multiple receive calls to get all the pieces of a single complete message and must also be prepared to receive multiple messages and partial messages at each call.

6.4 About Packets

In addition to IP and TCP headers, datacenter packets may contain additional headers. For example, virtual local area networks (VLANs) can be implemented by having a 4-byte VLAN header before the IP header. Cooperating routers deliver packets based on their VLAN header, with the

effect that packets from one virtual LAN can be prevented from reaching ports associated with other virtual LANs. This design allows multiple completely unrelated networks to use shared switching equipment. Packets without a VLAN header can be dropped by a router or sent to specific unsecured ports. Incoming packets with the wrong VLAN header for a particular port can be dropped. The goal is that each type of traffic is completely unable to observe any of the other traffic, even if some connected computers are spoofing their MAC and IP addresses to try to read, and even modify and forward, others' data. If the routers themselves operate correctly, this can give some level of security and privacy.

One use of VLANs is for a building-wide network with specific authorized machines (by MAC address) attached to specific router ports and using VLAN headers. An unauthorized machine connected to the network is not allowed to use any VLAN headers, and all it can see is a tiny default network consisting of itself and a gateway/authorization computer that may choose to stop all communication with the device, may convert it into an authorized node that can use VLANs, or may allow it to connect to an outside Internet port, thus supporting devices from guests visiting the building but otherwise allowing only limited access for those unauthorized machines.

Packets may also be encrypted. Enough initial information is left unencrypted to allow the packets to be routed, and then an encapsulation header is used to signal that the remaining bytes are to be passed on unchanged and uninterpreted by any routing mechanism. The encapsulated data can be encrypted in various ways by the sender and decrypted by the receiver. The encapsulation technique can also be used to carry byte streams that actually contain non-Internet bytes and use completely different routing protocols for some private network that connects different locations via encapsulated traffic sent over the regular Internet.

We will consider only unencapsulated packets for the rest of this book, since we are focusing on server-to-server network performance and not on all the possible ways to use the Internet.

6.5 About Remote Procedure Calls (RPCs)

Our lab experiments will use a form of remote procedure call. For a *local* procedure call, routine A calls some `Method` with arguments and gets back a return value, with all the code running on a single machine:

```
routine A {
    ...
    foo = Method(arguments);
    ...
}
```

For a *remote* procedure call, the idea is the same, but the `Method` (e.g., a C function) runs on a remote computer.

The `Method` name and arguments are passed to the remote server in a request message, and the return value is eventually passed back in a response message, as shown in Figure 6.5. The client and server programs are constructed with calls to an RPC library. Building, sending, and parsing the request and response messages is done by the library routines, implementing a particular RPC design. Non-blocking RPCs allow multiple RPC requests to be outstanding at once and allow responses to return out of order.

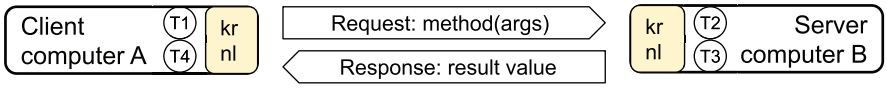


Figure 6.5 **A single RPC sending a request message and eventually receiving a response message; "krnl" is kernel code; T1-T4 are timestamps in user-mode code to send/receive RPC request and response messages.**

Each message is a network transmission. The request message goes from

- a user-mode client program on computer A at time T1 to
- kernel-mode code on A,
- over the network,
- to kernel-mode code on computer B,
- to a user-mode server program on B at time T2.

The response message travels in the opposite direction, at times T3 and T4. RPC latency is measured from the time T1 that the user-mode client program on A sends the request to the time T4 that the user-mode client program on A receives the response. When a response is delayed, the delay can be anywhere on the picture—request or response, user code or kernel code, machine A or machine B, send or receive network hardware. The four times T1..T4 help observe where the overall time went.

To examine the performance effects of network RPCs we will use timelines with events T1, T2, T3, and T4 indicating the RPC timing. We will draw individual RPCs as timelines with notches showing the times T1..T4, as shown in Figure 6.6. The notches do not take up much diagram space, but the human eye is quite good at picking them out, even when there are hundreds of RPC lines close together. The total RPC latency, as observed by the client user-mode program, is T4-T1. The total server time for the RPC is T3–T2.

Figure 6.6 **Diagram of one RPC, showing the four times. T1 to T2 is the time from client user-mode code sending an RPC request to server user-mode code receiving that request. T2 to T3 is the server time spent performing the request. T3 to T4 is the time from server user-mode code sending the RPC response to client user-mode code receiving that response. Times T1 and T4 are taken from the client CPU's time-of-day clock, while T2 and T3 are from the server CPU's time-of-day clock. The two clocks may be offset from each other by microseconds to milliseconds. We will deal with clock alignment in the next chapter. w1 is the time the client kernel-mode code sends the request to the network hardware ("w" for "wire"), and w3 is the time the server kernel-mode code sends the response to the network hardware.**

The return value from an RPC may be a single status number or may be thousands of bytes of data. It is convenient to always return both an overall status for the call (success, failure, specific error codes) and a possibly empty byte string of additional results.

Most datacenter software uses RPCs to send work between servers. For example, passing a paragraph of text to Google Translate via its web-page interface may send that paragraph to a load-balancing server that in turn forwards it to a least-busy translation server, which in turn may break the paragraph into sentences and send the individual sentences in parallel to a few dozen sentence servers that do sequences of multi-word phrase lookups in the source language and map into the best-score sequence out of many possible phrases in the target language. These results are then gathered back together by the translation server into a single translated paragraph.

6.6 Slop

The *slop*, or unidentified communication time, is (T4-T1) - (T3-T2) = (T2-T1) + (T4-T3). When the client RPC latency and the server time are nearly equal, the slop is small. When there are communication delays (usually in the kernel code on one machine or the other, not in the network hardware), the slop can be large. Figure 6.6 shows a large slop, with the overall RPC latency about 1.5x the server time. In Chapter 7, we will also subtract the estimated transmission time for request and response messages:

```
slop = (T4 - T1) - (T3 - T2) - requestTx - responseTx
```

When the slop is large, that means that there is significant delay somewhere between the two communicating user-mode programs. In Chapter 15 we will introduce recording the w1 and w3 times of RPC headers on the wire, shown in gray in Figure 6.6. They indicate here that request and response messages hit the wire almost immediately when sent, so the long delays are in kernel code on the receiving end.

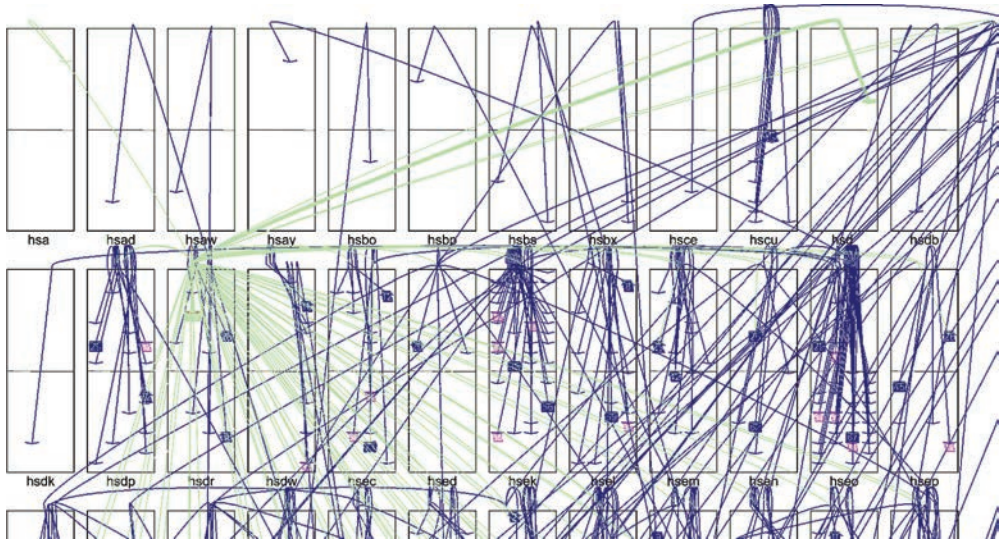


Figure 6.7 **Two-level RPC call tree for a single web search, farming out one search to about 2,000 partial searches of different parts of a web index. Each rectangle represents a rack of ~50 server machines. The light green arcs show about 100 top-level RPCs from one machine in rack "hsdr" to 100 others. The dark blue arcs show about 20 second-level RPCs from each of those 100 to a total of about 2000 servers.**

To complete the RPC picture, Figure 6.7 shows a two-level call tree of RPCs doing a single web search. The top-level ~100 light green RPC arcs coming from the top of the rack labeled "hsdr" are all done in parallel, and all the groups of second-level ~20 dark blue RPC arcs are also done in parallel, quickly spreading the work across about 2,000 servers. Each leaf in the call tree does a portion of the search, and those partial results are combined when all parallel RPCs in a group have returned.

6.7 Observing Network Traffic

In this chapter we will observe and measure some simple RPCs. In contrast to observing local CPU, memory, and disk activity, it takes two connected machines and two sets of software to observe network traffic. Rather than just observing isolated packets, we will observe an RPC system that has client software, server software, multi-packet RPC request and response messages, multiple server threads, and overlapped client calls. As usual, we wish to observe in enough detail to detect anomalous dynamics.

Figure 6.8 shows one example of such dynamics, captured by RPC logs and Dapper [Sigelman 2010]. Using the style of our single-RPC diagram from Figure 6.6, the notched lines in Figure 6.8 show the time layout of 93 parallel RPCs, similar to the top-level RPCs shown as timeless light green arcs in Figure 6.7.

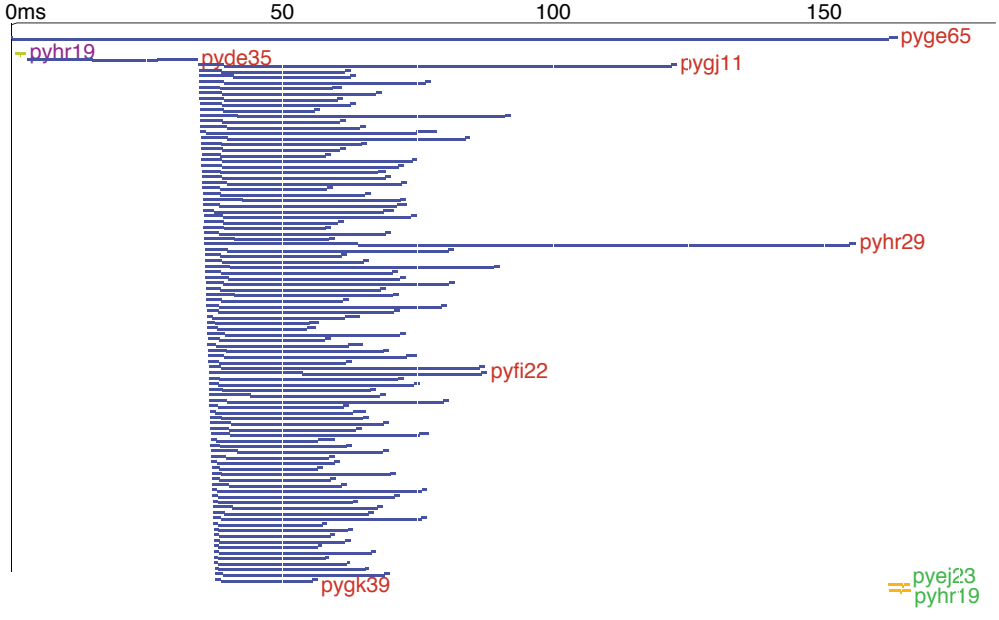


Figure 6.8 **Diagram of ~100 RPCs at top level of a single web-search RPC. The "pyxxxx" notations to the right of each line are individual server names.**

The very top notched blue line labeled pyge65 in Figure 6.8 shows an incoming single RPC requesting a single web search on server pyge65. It takes about 160 msec total. Underneath it are the outbound sub-RPCs that this initial call spawns, directed at other servers. You can see the sub-RPC transaction latency variation and can see that the 99th percentile slowest parallel RPC on pyhr29 determines the response time of the overall web-search RPC. You can also see that understanding and removing the sources of long latency can speed up this example by about a factor of 2, from 160 msec total to about 80 msec.

Look at the spawned RPCs. First, at the far upper left there is a short barely visible call to server pyhr19 (yellow line just under the "m" in "0ms") to check for a previously cached immediate answer. Using cached previous search answers noticeably speeds up identical searches. Then a blue call to pyde35 is a *canary request* [Dean 2010]. Only when that returns successfully, i.e., without crashing pyde35, are the other RPCs done. If you have a request that hits a code bug that crashes a server, *and you will*, the canary strategy results in crashing just one server instead of thousands. In Figure 6.8 the canary returns, so the 90-odd parallel calls to pygj11 .. pygk39 are started. Not shown are the 20-odd RPCs that each of these spawn in turn, about 2,000 in total, corresponding to the dark blue arcs in Figure 6.7.

Only when pyhr29 returns, the slowest of these parallel calls, does the initial web-search RPC complete. At the very lower right are two parallel calls to update duplicate cached results on pyej23 and on pyhr19 (yellow lines). These actually occur after the initial RPC completes. The vertical white line at 50 msec is just a time grid.

If you look carefully at the canary call to pyde35, you will notice that the request message takes over 10 msec to go from client user-mode code to server user-mode code, and the response message also takes over 10 msec to go from server to client. This slop is much longer than the slop time for most of the subsequent RPCs, so we have our first unexpected source of excess latency. For datacenter networks within a single building, the delay through the routers of the hardware switching fabric rarely exceeds 20 usec. So a delay that is 500x longer than that can only be a software, not hardware, delay, somewhere on the client or server in either user code or kernel code. We examine such delays in Part IV.

If you look carefully, the 93 parallel calls do not all start at exactly the same time—there is a slight tilt to the nearly vertical left edge. Their start times increment by about 6 usec each, reflecting the CPU time to create and send each RPC. The leftmost notch on each line shows that almost all the RPC requests arrive at their corresponding server program fairly quickly, except the calls to pyhr29 and pyfi22, which take over 20 msec to arrive. This is another latency mystery to be resolved.

The rightmost notch on each line shows that almost all the RPC responses are sent soon before they arrive at the client program, so there is no latency mystery for those.

Initially, however, we would be more interested in the exceptionally slow response times of pyhr29 and pygj11, since they delay the overall response time of the initial RPC by about 70 msec. Understanding those delays requires observing what is happening on each of those CPUs, applying our observation tools and thought to each in turn. The same kind of transaction log files that were used to create Figure 6.8 can be used on the logs from pyhr29 and pygj11 to see the dynamics of their delays. The general pattern is: observe to focus on the big issues and ignore the inconsequential artifacts, examine the important artifacts in more detail, resolve them, and then repeat.

The good news is that our picture of the RPC activity on just one machine has revealed two message-delivery latency mysteries and has pinpointed exactly the other two machines and time of day to the microsecond that contribute to overall slow response time for this one web search. Looking at multiple such web searches over a few tens of seconds will reveal whether pyhr29 and pygj11 are always slow or just happened to be slow for this one observation.

Our goal in this chapter is to capture enough information about each RPC to be able to draw diagrams like Figure 6.8 and then use those to track down root causes for delays. In later chapters, especially Chapter 26, we will add tools for observing the underlying reasons for delay(s) that our RPC diagrams reveal.

But before exploring the observation of network dynamics, we need to describe the sample "database" RPC system in a little more detail.

6.8 Sample RPC Message Definition

Local procedure calls have relatively simple dynamics—on a given CPU core, procedure A calls procedure B, and that CPU core then executes instructions in B; no further instructions in A execute until B returns. Local procedure calls may be nested, with A calling B, which in turn calls C. But these all execute sequentially on a single CPU core. We can observe a complete local call tree by capturing the entry and exit times of each procedure. Call nesting is implied by nested entry/exit times. In a multiple-core, multi-threaded-program environment, multiple local calls to B can occur simultaneously, but they are from different callers executing on different software threads possibly using different CPU cores.

Remote procedure calls are more complicated. Unlike a subroutine call, the transmission of the request message from client machine A to server machine B is not instantaneous, nor is the transmission of the response message back from B to A. Since these messages use shared network resources, other network traffic may delay them, so we at least want to capture send/receive times for each message.

As shown in Figure 6.8, an RPC may be *non-blocking*—the caller A may proceed with additional execution in parallel with B and issue additional parallel RPCs to C, D, E, etc. Eventual response messages from B, C, D, E, ... arrive at A asynchronously and not necessarily in order. To match up multiple request and response pairs in the RPC library code, each outstanding RPC is given a unique ID, and that is included in its request and response messages. To match up one RPC with any sub-RPCs that it does, each sub-RPC also includes the RPC ID of its parent; this allows us to reconstruct entire call trees.

A caller may wait for all its RPC responses before finishing, or it may finish early as in Figure 6.8 (pyge65 at the very top right returns before the calls to pyej23 and pyhr19 at the very bottom right). If a network link goes down or a server crashes, some responses may never arrive; the caller needs to detect and deal with this rather than waiting forever.

During the time that A is waiting for a response from B, other clients on the same or different machines may also be sending RPCs to B, and B may be working on those and not on A's. If that happens, a sub-RPC from B to some other server Z may be part of the work for A or for any other client of B. The parent ID shows the proper association.

> If A calls B with RPCID 1234 in the request message and B subsequently calls Z on behalf of A's request, the call to Z would have parent ID 1234 and its own RPCID of perhaps 5678.

All these complications happen fairly often in a large datacenter environment.

We can observe the dynamics of a complete remote call tree only by capturing the caller/callee pairs and send/receive times for each request and response message and explicitly recording the parent caller for all nested RPCs. Most of this information must be transmitted between machines in each of the request and response messages.

For our sample RPC system, each request or response message starts with an RPC marker followed by an RPC header followed optionally by a byte string that contains the argument values for a request or the result values for a response, as shown in Figure 6.9. Each complete message is broken up into the payload data carried in one or more TCP/IP packets. We will focus in the rest of this chapter on complete messages instead of individual packets.

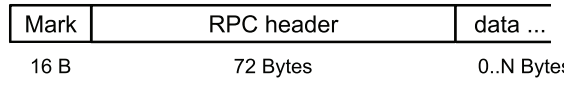


Figure 6.9 **Overall structure of a request or response message in our sample RPC design**

The 16-byte RPC marker, as shown in Figure 6.10, serves several purposes: delimiting messages, defining variable lengths, and sanity check.

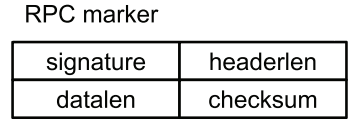


Figure 6.10 **RPC marker of 16 bytes**

The signature is a fixed 32-bit value. This allows a quick check that the subsequent bytes could begin an RPC message and are not something else. If somehow a TCP connection gets out of sync, it also allows scanning forward until a signature is found as a way to resynchronize. (This is not necessarily a good idea; it may be better to drop the connection and force a clean restart.) In Chapter 15, we use the signature field to filter packets that appear to be the beginning of an RPC message, recording KUtrace entries for each.

The 32-bit headerlen field gives the byte length of the following RPC header, whose size will likely vary over several months or years in a real datacenter as the RPC library is updated and expanded. To improve validity checking, headerlen values are required to be less than 2**12. For our sample RPC design, headerlen is always 72.

The 32-bit datalen field gives the byte length of the optional argument or result byte string that follows the RPC header. A length of 0 indicates no string. To improve validity checking and to make huge messages invalid, datalen values are required to be less than 2**24. The two length fields allow an RPC library to break a message into its variable-length pieces.

Finally, the 32-bit checksum field is a simple arithmetic function of the previous three fields, allowing a robust sanity check that the marker and subsequent bytes are highly likely to be the start of a valid RPC message.

The RPC marker is designed to be part of a complete network message but is not visible to the callers of the RPC library software. Those callers deal only with the RPC header and data string.

The RPC header, shown in Figure 6.11, has all the information to describe a single RPC request or response message. Fields are initialized to zero and are filled in incrementally by RPC library as an RPC is processed. For example, T1 and the first L are filled in by the RPC library when an RPC request message is about to be sent by a client program. The second L is filled in when an RPC response message is about to be sent by a server program, and T4 is not filled in until the RPC response message is received by the client program.

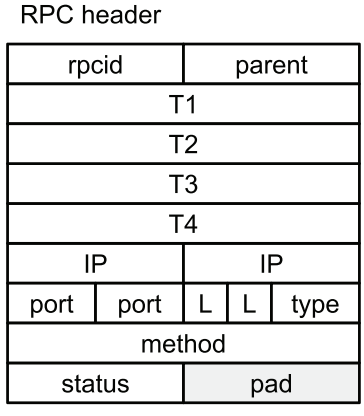


Figure 6.11 **RPC header of 72 bytes.**

Briefly, the naturally aligned fields are

- **RPCID** 32 bits, containing a unique ID number for each outstanding request.
- **Parent ID** 32 bits, containing the RPCID of the request that spawned the current request.
- **T1..T4** 64-bit wall-clock timestamps with microsecond resolution, giving respectively the request send time, request receive time, response send time, and response receive time; T1 and T4 are based on the client machine's time-of-day clock; T2 and T3 are based on the server machine's time-of-day clock.
- **IP** 32 bits and **port** 16 bits, giving the client and server machines' TCP/IP addresses,
- **L L** 8 bits each, giving the logarithm of the byte lengths of request and response messages.
- Message **type** 16 bits, to indicate request or response or other types of message.

- **Method** 64 bits (8 bytes), ASCII name of the routine being called, zero padded.
- **Status** 32 bits, return-value status indicating success, failure, or specific error number.
- **Padding** 32 bits, to make the header a multiple of eight bytes in total length.

The sizes of the RPC header fields are somewhat arbitrary; different sizes would work equally well. Reducing the byte lengths to logarithms is just an example of trading resolution (e.g., within 10%) for space.

This header format is somewhat less flexible than those used in real datacenters, but is sufficient for our sample RPC work.

6.9 Sample Logging Design

The client-server programs described in the coming text each write a log file of all the RPCs they process. This logging is our designed-in observability for the dynamics of the RPC system. As such, it is important that the logging is not so slow or bulky that it consumes significant resources or distorts the performance of the underlying service.

Our sample design target is to be able to process and therefore log up to 10,000 RPCs per second with little overhead. This is the right order of magnitude for a real datacenter service.

Back-of-the-envelope: If each log entry is 1,000 bytes, logging 10,000 RPC/sec would write 10MB/sec to a log file, or 864GB/day per service, with multiple services running on each server. This quickly gets bulky and can also consume significant bandwidth to a disk holding multiple log files. Each service would nearly fill up a 1TB disk every day, likely requiring multiple disk drives just for logging.

Instead, if each log entry is about 100 bytes, one service would log about 1MB/sec, and total about 86 GB/day. This is still a bit bulky, but several services writing 1MB/sec to a logging disk is an easily sustained rate, and those services would need just a single 1TB disk to hold a day's worth of log entries—a manageable amount.

For slower-rate services that handle only about 1,000 RPCs per second, we could roughly afford 1,000 bytes per log entry, although most logging does not need so much data per RPC. Do the back-of-the-envelope arithmetic at design time to document affordable limits on the logging overhead.

For our sample RPC design, the binary log format is just a copy of the current RPC header, with the full data length from the RPC mark moved to just in front of the data, and the data itself truncated or zero-extended to 24 bytes, as shown in Figure 6.12. Each log entry is thus exactly 96 bytes.

Log-system performance is a key consideration. Truncating the data keeps each log entry size bounded, and it also provides a bit of privacy for possible user-supplied data such as an email message. Including even a little of the data, though, helps identify what is going on when there is unusual latency. Recording log entries as binary fields instead of printed ASCII values saves file space and also saves CPU time for formatting all the numbers. This approach substantially reduces the logging overhead. Conversion to readable ASCII can be done by postprocessing binary log files as needed later.

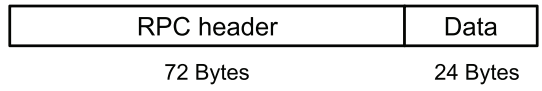


Figure 6.12 **Sample log entry format, 96 bytes total**

Each server program will write a local binary file of these log entries as it runs, and each client program will write its own local binary file of log entries. For an overall service running on 2,000 machines, the server programs would write 2,000 local log files on those machines. Scattered clients would write local log files on their own machines.

In a large datacenter, all programs would periodically close their log files and open new ones. A background service would gather closed log files into a single place or into a distributed file system so that they could be postprocessed efficiently. To conserve space, most logs would be thrown away after a few days. There is little point keeping data if no one will ever read it again. For our sample environment, none of that log management is done: we just deal with multiple local log files written locally on individual servers.

6.10 Sample Client-Server System Using RPCs

The supplied `server4.cc` and `client4.cc` implement a sample RPC system with logging. In addition, the supplied `dumplogfile4.cc` turns binary logs into JSON-formatted ASCII so you can see what they contain and so that they can be displayed easily.

The service provided by these programs is an in-memory key-value store (a simple database). The server program accepts RPCs that read or write key-value pairs in RAM, while the client sends such requests. The server-implemented methods are

- **Ping**(data), returns a copy of data with no key-value action
- **Write**(key, value), remembers the pair
- **Read**(key), returns value
- **Chksum**(key), returns an eight-byte checksum of the value
- **Delete**(key), removes key and its value
- **Stats**(), returns some statistics on server usage
- **Reset**(), erases all key-value pairs
- **Quit**(), stops the server and all its threads

This pair of programs should allow you to get into trouble in myriad ways. Sending a burst of 100 values of 1MB each will saturate a sample server's 1Gb/sec Ethernet for at least a second. Doing two such actions independently between different machines should overload a sample four-port switch. Sending a burst of 100,000 one-byte values will saturate CPUs quite nicely and also clog up the logging system. Overlapping a heavy-duty burst with some more modest work will likely interfere with the modest work.

6.11 Sample Server Program

The server program shown in Figure 6.13 continuously accepts RPC request messages, processes them, and sends response messages. It usually runs with multiple threads handling independent RPCs applied to a single shared database.

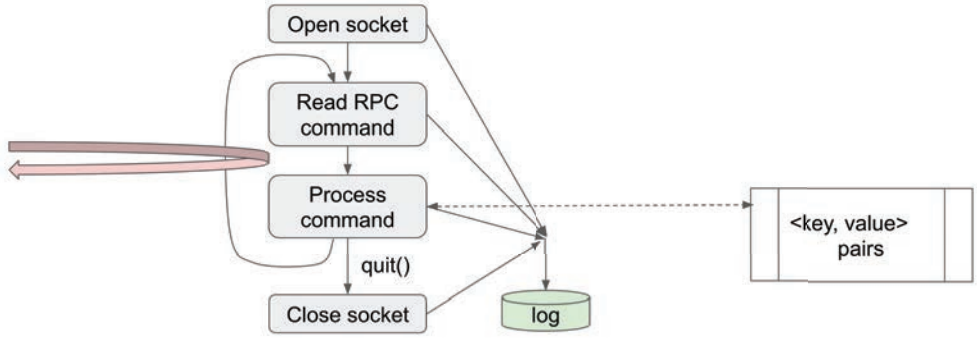


Figure 6.13 **Sample server program**

The program, `server4.cc`, takes two command-line arguments specifying the range of ports to listen on. For each port, it forks a dedicated listening thread. Each such thread opens a TCP/IP socket on its port and waits for RPCs, which it then executes sequentially as they arrive. The default behavior is use four sequential ports, 12345..12348, and to fork four corresponding threads.

As a safety move, each launch of `server4` will self-destruct after four minutes, to protect against runaway or zombie programs, even if `server4` is launched in the background via the command-line ampersand.

Multiple copies of `server4` can be launched, so long as they use non-overlapping port numbers. Copies of `server4` can run on multiple server machines. Nothing in the simple design except chance prevents multiple people from launching interfering runs at the same time.

Keys are byte strings restricted to less than 256 bytes, while values are byte strings restricted to less than 1.25MB (i.e., 5 * 256 * 1024 bytes). Total RAM storage space is restricted to be less than 200MB.

The database of key-value pairs is a C++ map of strings. It is shared across all server execution threads, so each operation that touches the database takes out a simple spinlock before accessing the data. By design, this is a somewhat flawed approach—everything works, but there can be severe blocking dynamics, which we will shortly observe.

6.12 Spinlocks

The spinlock protecting a software *critical section* is our fifth shared resource, along with the four hardware resources CPU, memory, disk, and network. There are many forms of software locks, with spinlock the simplest—whenever a thread cannot acquire the lock for a critical section of

code, it simply loops (spins) trying over and over to acquire the lock, until eventually other threads free the lock and the subject thread successfully grabs it. Chapter 27 discusses locks in more detail.

The sample server code defines a C++ `SpinLock` class, which acquires a spinlock in its constructor and frees the lock in its destructor. Thus the code pattern

```
LockAndHist some_lock_name;
  ...
{
  SpinLock sp(some_lock_name);
  <critical section code here>
  ...
}
```

makes the inner block a critical section that can be executed by only one thread at a time. The C++ constructor/destructor mechanism for `SpinLock` guarantees to acquire the lock `some_lock_name` upon entry to the block and to release it upon exit from the block, even for an unexpected or exception exit. This design completely removes one source of programming error—processes that sometimes fail to release a lock.

The spinlock implementation also defines a small histogram to record lock acquisition times. This is another piece of designed-in observability. A common issue with software locks is that under some circumstances a program thread has to wait much too long to acquire a lock, resulting in long transaction latency on one thread whenever *another thread* holds a lock too long. A small histogram of lock-acquire times for each lock can tell you the normal time taken to acquire a contended lock and also can show how many much-longer times occur. If there are no long acquisition times for a given lock, then lock-waiting is not a cause for a long transaction latency, and you can look elsewhere.

The locking pseudocode looks like this:

```
start = __rdtsc()
  test-and-set loop to get lock
stop = __rdtsc()
elapsed_usec = (stop - start) / cyclesperusec
hist[Floorlg(elapsed_usec)]++
```

where `rdtsc` reads the x86 cycle counter and `Floorlg(x)` takes floor(log2(x)), returning 0..31 for a 32-bit unsigned int `x`. The variable `hist` is a small array of counts. Logarithm base 2 is sufficient resolution to put long and short acquisition delays into different count buckets. The `stats` command mentioned earlier returns this histogram array.

6.13 Sample Client Program

The client program, shown in Figure 6.14, takes command-line arguments and sends one or more RPCs to a specified server and port, repeated and spaced out in time in a stylized way.

Multiple instances of client4 can be launched in the background so that they overlap in time on a single machine, and of course multiple instances can be run from several different machines. Instances of server4 and client4 can also run on the same machine, but will not use the network for local communication; for this degenerate case, the kernel network code moves message bytes in RAM.

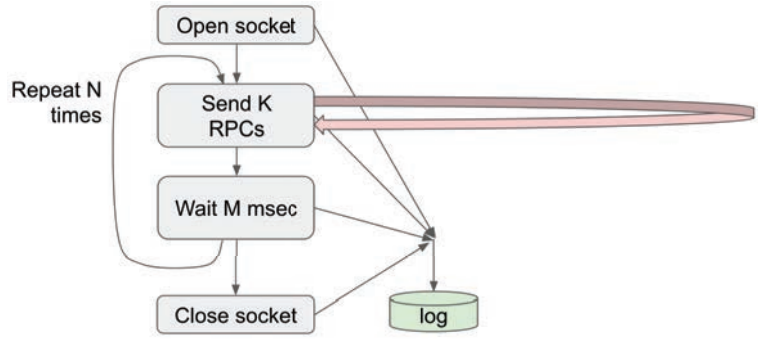


Figure 6.14 **Sample client program**

The command-line arguments to client4 allow specifying Rep repetitions of the pair of actions

<send K RPCs, wait M msec>

The first RPC of a burst of K is specified by its method and initial key/value data. Either field can be padded with pseudorandom data to a specified byte length. Subsequent RPCs in a burst keep the method and base strings but supply possibly incremented keys/values. Within a burst, each RPC is sent as soon as the previous one returns a response, but not before.

The client4 program takes a set of command-line parameters that form a little language:

```
./client4 server port
    [-rep number] [-k number] [-waitms number] [-seed1]
    [-verbose] command
    [-key "keybase" [+] [padlen]]
    [-value "valuebase" [+] [padlen]]
```

The write, read, and delete commands require a key, and the ping and write commands require a value.

`-rep`	Says to repeat the outer loop some number of times.
`-k`	Says to repeat the command some number of times (the inner loop) and then wait before continuing the outer loop.
`-waitms`	Says how long to wait (in milliseconds) after each burst of k commands.
`-seed1`	Says to seed the random number generator for padding bytes to exactly the value 1, allowing reproducible pseudorandom values.
`-verbose`	Prints a little about each request and response message.
`-key "keybase" [+] [padlen]`	Specifies a base character string that is optionally incremented and optionally padded with random characters. In the presence of -rep and -k repetitions, "+" indicates incrementing the base string, and padlen gives the padded length.
`-value "valuebase" [+] [padlen]`	Uses the same algorithm.

For example,

```
./client4 target_server 12345 -k 5 ping -value "vvvvv" + 10
```

sends five ping commands to target_server:port, with value strings such as

```
vvvvv_0u5j
vvvvw_trce
vvvvx_qxol
vvvvy_1bv3
vvvvz_dg1w
```

where the + specifies incrementing `v w x y z` in the base string, and the `10` specifies padding with random characters out to 10 characters total. Incrementing increases the low character of the base string by 1, wrapping 9 to 0, z to a, and Z to A, carrying into higher character places as needed (the next base value above would be vvv**wa**). Incrementing is most useful for keys and padding is most useful for values.

The individual commands are defined in a little more detail here.

ping [-value "valuebase" [+] [padlen]]

Sends an RPC request to the specified server:port containing RPC marker, RPC header, and optionally RPC data containing the specified value. The server responds with the same data.

write -key "keybase" [+] [padlen] -value "valuebase" [+] [padlen]

Sends an RPC request to the specified server:port containing RPC marker, RPC header, and RPC data containing the specified <key, value> pair. The server saves each <key, value> pair and responds with a status code, typically SUCCESS.

read -key "keybase" [+] [padlen]

Sends an RPC request to the specified server:port containing RPC marker, RPC header, and RPC data containing the specified key. The server responds with the matching value and a status code, typically SUCCESS.

delete -key "keybase" [+] [padlen]

Sends an RPC request to the specified server:port containing RPC marker, RPC header, and RPC data containing the specified key. The server deletes each matching <key, value> pair and responds with a status code, typically `SUCCESS`.

stats

Sends an RPC request to the specified server:port containing RPC marker, RPC header, with no RPC data. The server responds with an arbitrary status string and a status code, typically SUCCESS. For server4, the status string is a text version of the spinlock histogram described earlier—32 counts separated by single spaces.

reset

Sends an RPC request to the specified server:port containing RPC marker, RPC header, with no RPC data. The server removes all its <key, value> pairs and responds with a status code, typically SUCCESS.

quit

Sends an RPC request to the specified server:port containing RPC marker, RPC header, with no RPC data. The server responds with a status code, typically SUCCESS, and exits immediately afterward.

The client4 program prints the observed round-trip time for the first 20 RPCs it issues. At the end, it prints a log2 histogram of those times, plus total RPCs, total msec elapsed, total MB transmitted and received, and RPC messages transmitted and received per second. In a datacenter system, this is the kind of information that would be displayed on a dashboard web page.

6.14 Measuring One Sample Client-Server RPC

Recall our thought framework from Chapter 1. In thinking about performance issues, we will follow the programmer's discipline of first estimating how long some work should take, then observing how long it actually does take, and then reasoning about any differences. Figure 6.15 shows this framework again.

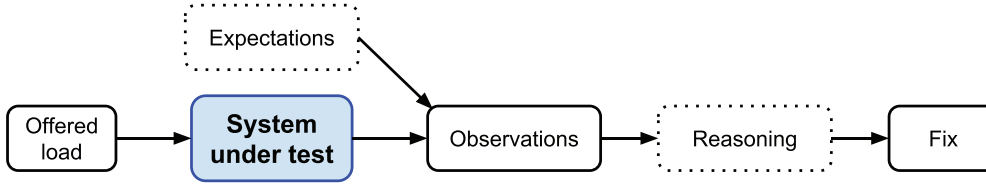


Figure 6.15 **Framework for examining the performance of complex software**

Consider the offered load of sending a single **write** RPC from client program to server program, with a 5-byte key and 1,000,000-byte value. What do we expect the timing to look like on our sample server configuration with a 1Gb/sec network?

Remember that 1 Gb/sec is roughly 100 MB/sec when we convert from bits to bytes and include some overhead, or about 100 bytes per microsecond. So sending ~1,000,000 bytes of RPC request should take about 10,000 usec (10 msec) across the wire. On the server side, creating a string of 1,000,000 bytes and putting it into a C++ map should take only about 100 usec or so if the main memory can transfer 10GB/sec. A short reply of 100 bytes should take about 1 usec on the wire plus some small software overhead. Overall, we might expect something like Figure 6.16 when we sketch out a timeline.

Figure 6.16 **Expected RPC timing sketch for Write() of 1MB of data**

This sketch is somewhat distorted, but you get the idea—a long time to transmit 1MB in the request message, 100x less time to put it into the key/value store, and 100x less time again to transmit the response message. Let's see what happens.

Compile server4.cc and run it with no arguments to get the default configuration of listening on four ports.

```
./server4
```

Compile client4.cc and run it on a different machine with these arguments:

```
./client4 target_server 12345 write -key "kkkkk" \
  -value "vvvvv" 1000000
./client4 target_server 12345 quit
```

Compile dumplogfile4.cc and run it on the client, pointing to the 1MB write log file written by client4.

```
./dumplogfile4 client4_20190420_145721_dclab-1_10479.log \
  "Write 1MB" \
   >client4_20190420_145721_dclab-1_10479.json
```

Compile makeself.cc and run it on the client, pointing to the previous JSON file.

```
./makeself client4_20190420_145721_dclab-1_10479.json \
  show_rpc_2019.html \
  > client4_20190420_145721_dclab-1_10479.html
```

Display the resulting HTML file.

```
google-chrome client4_20190420_145721_dclab-1_10479.html
```

6.15 Postprocessing RPC Logs

The programs `client4.cc` and `server4.cc` write binary log files of 96-byte records as described earlier.

The program `dumplogfile4.cc` reads these log files and turns them into JSON files with the timestamps and other information turned into ASCII text. The JSON file has a stylized header that contains among other things the start minute of the log records, the title from the second command-line argument ("Write 1MB" previously), and some axis labels. This header is followed by lines of text for the log records. By default, only log records for receipt of a response are included, i.e., just those records that describe a full round-trip transaction. The -all flag includes all records.

The program makeself.cc reads a JSON file and writes an HTML file based on a template, incorporating the JSON information. This is the same makeself.cc program we encountered in Chapter 5, but with a different template file, `show_rpc.html`.

The displayed HTML, such as Figure 6.17, can be panned and zoomed via mouse drag and mouse wheel, respectively. The lower-left red dot resets the display. The [Rel. 0] button at the top switches between showing multiple RPCs by wall-clock- time and showing them all starting at (relative to) time zero.

6.16 Observations

On a sample server pair of machines, I measured 9.972 msec to send the Write() request from client to server, 1.118 msec to process it on the server, and 10 usec to send the response back to the client. My estimates of the first and last times were pretty reasonable, but my processing time estimate was 10x too low (see Table 6.1).

Table 6.1 **Estimated RPC Elapsed Time vs. Measured**

Action	Estimate	Actual
Send 1MB request	10 msec	9.972 msec
Process request	100 usec	1118 usec
Send 100 byte response	2 usec	10 usec

Based on the client-side log, Figure 6.17 shows 10 such RPCs lined up on time across about 120 msec, using the style of Figure 6.8. The hollow part of the lines (more precisely, the white overlaid lines) show approximately the message transmission time for the 1MB client => server messages. Among other things, you can see the 1–2 msec gap on the client from the end of one RPC to the beginning of the next one (oval around the first gap).

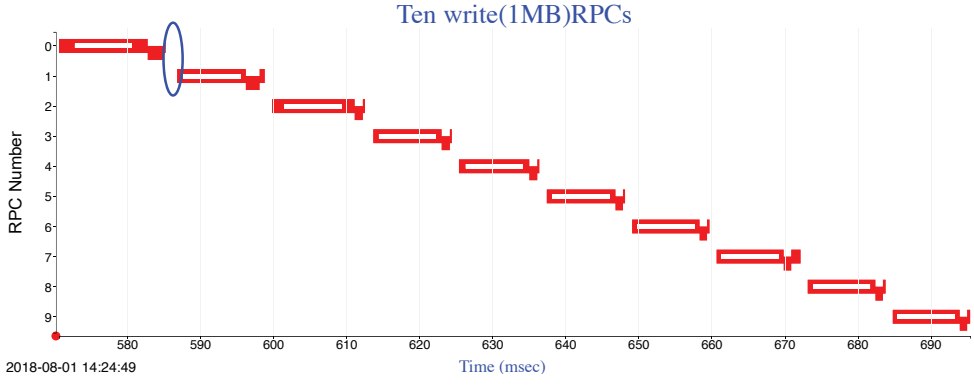


Figure 6.17 **Ten RPCs, each sending 1MB of data. The oval highlights the 1–2 msec gap between successive RPCs.**

Figure 6.18 shows the same 10 RPCs, but this time they are aligned to a common starting point, so the relative times of the different RPCs are easily compared. Now it is clearer that the first few RPCs take longer, and then the timing settles down a bit. It is also clearer that number seven has an extra delay in getting its response message back to the client.

The request time from client to server is about 9 msec, as we estimated, but the time processing on the server is about 1 msec, longer than the 100 usec per 1MB that we estimated. Why was our estimate off?

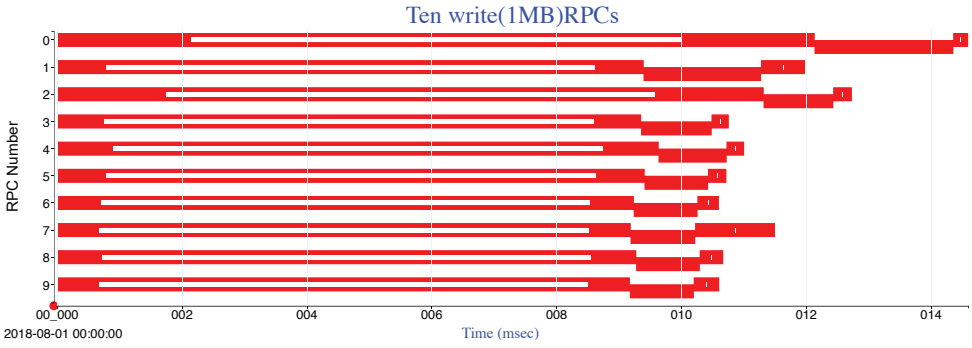


Figure 6.18 **The same 10 RPCs, but this time all with the same start time of zero**

Copying each 1MB message from network card to kernel buffer, then kernel buffer to user buffer, then user buffer to separate key and value strings, and then value string to map entry, each one MB in our sample server code is read or written to memory about eight times, not just once. This boosts our estimate to about 800 usec for all those copies, close to the observed 1 msec. To complete the picture, the response time from server to client is about 350 usec, a little longer than I would have expected but not terrible. Estimating ahead of time makes it easy to spot discrepancies.

6.17 Summary

In this chapter we introduced a sample remote procedure call database system and ran it on two networked computers, recording the times at which each RPC message is sent and received. The resulting notched-line diagrams show the measured time, the approximate transmission time for each message, the spacing between successive RPCs, and the relative timing of multiple similar RPCs. The underlying logging is sustainable at a rate of 10,000 RPCs per second per service.

In the next chapter, we will look in more detail at multiple overlapped RPCs on multiple clients and servers, at locking within transactions, and at time-aligning the clocks on multiple machines. We will expand the sample in-memory database to be an equally simple disk database. Later in the book we will examine the slow parts of these RPCs, including identifying the delays in message transmission and the delays in RPC processing.

- To understand the dynamics of multiple remote procedure calls, it is absolutely necessary to design in observation hooks, including at least RPC IDs, send and receive times, and byte lengths.
- These hooks must have low enough overhead to be useful under heavy live load.
- Creating RPC traffic requires at least one client and one server program running on different machines.
- A little stylized language lets us build a client to generate useful sequences of RPCs.

- Data structures for the RPC format across a network are part of the overall RPC design, and these must include observation metadata in the messages sent across a network, along with the actual operation and its data. This metadata information can fit in about 100 bytes.
- Logging to disk the timestamps and other metadata for every incoming and outgoing RPC message lets us observe a complete picture of where all the time went in an RPC call tree, and also a complete picture of all the *other* RPCs whose processing overlaps with and therefore possibly interferes with one of interest.
- Viewing the log data by wall-clock time lets us see delays within and between successive RPCs, and also lets us see overlapping of RPCs.
- Viewing the log data with all RPCs of a given kind starting at time zero lets us see differences among similar RPCs, and in particular lets us see what is different about slow ones vs. normal ones.
- Estimating what we expect to see makes it easy to spot discrepancies.

Exercises

Consider this work:

1. Send 10 ping messages of 100KB each.
2. Send 10 writes of 1MB of random data for keys kkkkk, kkkkl, kkkkm, ..., kkkkt.
3. Send 10 matching reads of 1MB from the same 10 keys.
4. Finally, send a quit command.

Draw yourself a little sketch of what you expect to see in the RPC timings.

Now run the `server4` program on one sample server and the `client4` program on another, sequentially sending commands for the previous sequences. Run the dumplogfile4 program and makeself program against the first three client log files and display the actual results.

You likely will find that the two servers' wall-clock times differ by a few milliseconds, which may be enough to make the HTML display look odd, if the send time for a message is timestamped after the receipt time. We will look at time alignment in the next chapter. In the meantime, you might consider hand-editing the JSON files to adjust T2 and T3 to be between T1 and T4. This is optional, but doing so will give you some insight about what your Chapter 7 program will need to do.

6.1 How long, in milliseconds, did you estimate for the ping requests and their response message transmissions? How long do they actually take? Briefly comment on the difference.

6.2 How long, in milliseconds, did you estimate for the write requests and their response message transmissions? How long do they actually take? Briefly comment on the difference.

6.3 How long, in milliseconds, did you estimate for the read requests and their response message transmissions? How long do they actually take? Briefly comment on the difference.

This page intentionally left blank

Index

A

B

C

D

G

H

M

P

Q

R

S

T

U

V

W

X

Y

Z