The Art of Immutable Architecture

Theory and Practice of Data Management in Distributed Systems

—

Second Edition

—

Michael L. Perry

Apress®

The Art of Immutable Architecture

Theory and Practice of Data Management in Distributed Systems

Second Edition

Michael L. Perry

Apress®

The Art of Immutable Architecture: Theory and Practice of Data Management in Distributed Systems, Second Edition

Michael L. Perry
Allen, TX, USA

ISBN-13 (pbk): 979-8-8688-0287-4 ISBN-13 (electronic): 979-8-8688-0288-1
https://doi.org/10.1007/979-8-8688-0288-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jessica Vakili
Development Editor: Laura Berendson
Coordinating Editor: Jessica Vakili

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To Jenny. I still wouldn't change a thing.

Table of Contents

About the Author

Michael L. Perry is Director of Consulting at Improving, where he applies his love of software mathematics to benefit his clients. He has built upon the works of mathematicians such as Marc Shapiro, Pat Helland, and Leslie Lamport to develop a mathematical system for software development. He has captured this system in the Jinaga open source project. Michael often presents on math and software at events and online. You can find out more at http://michaelperry.net.

Acknowledgments

As Tolkien reminds us, it's dangerous business going out your door. The first step onto the road that has led to this book being in your hands was a shaky one. I was building my first distributed system and making all of the rookie mistakes. Fortunately, I had good friends to make those mistakes with.

Thank you, Russell Elledge and Jerry Feris, for fumbling alongside me. Together, we the Three Amigos learned all the wrong ways to use TCP/IP and SOAP. Who knew that the three-way handshake was not sufficient to guarantee delivery?

Although those first attempts were rough, we started to figure things out. Russell has been my constant co-conspirator, sounding board, and critic throughout this journey. I need to thank you also for introducing me to Chris Gould, who gave us both the freedom to apply what we had learned since that fateful first attempt. His support enabled us to build just the right solution on a mathematically sound foundation. It was the success of that project that gave me the final confirmation that these concepts can be taught.

To my constant collaborator and enthusiast Jan Verhaegen, thank you for encouraging me to package the system in one comprehensive reference. Thanks also for the motivation to put the system into practice with Jinaga. We are going to build great things together.

A huge thank you goes out to Sean Whitesell for years of support, encouragement, and discussion. You always ask the best questions. Just as importantly, you are skilled at bringing people together. Thank you for building the community that helped me practice communicating the ideas that ended up in this book. And thank you especially for making the final connection to get this project started.

It was also Sean who introduced me to Floyd May. Floyd, you are such a deep thinker in technology, interpersonal relationships, and business. You have challenged me to become a better communicator. I cannot wait to see where your feet sweep you off to.

To all of my friends at Improving: Cori Drew, Harold Pulcher, Barry Forrest, Ben Kennedy, David Vibbert, David Belcher, David O'Hara … all the Daves. We have grown so much together. I remember the first time I met each of you, and all of the things we learned since then. Thanks especially to Tim Rayburn for helping me grow as a speaker,

as an Improver, and as a leader. And now that I know that Devlin Liles has read this far into the acknowledgments, I guess it's OK to tell him that I think he's the most brilliant person in the company. He keeps his ego in check nonetheless.

A special thanks to Joan Murray at Apress for believing in this project, Jill Balzano for seeing me through my first publishing experience, and Shonmirin P.A. for staying with me through the second edition. Thanks, also, to Sander Mak for all of the challenging and insightful remarks. And to Jeff Doolittle for joining the fellowship and sharing the concepts with so many. You all made this process the most fun I've had doing the most difficult job.

And finally, my most sincere gratitude to my family. Dad, you inspired me to build software. You provided not only the Apple II and IBM that saw me through high school but also the introduction to the first person I saw making a living doing what I love. You kept the *Nibble* and *Byte* magazines coming in to quench my thirst and eventually to inspire me to write about what I've learned. I am the man I am because of you.

To Jenny. You have always believed in me. You are my partner and my reason.

And Kaela. You make me proud. I am so happy we finished this project together.

The road goes ever on and on.

Introduction

It was 2001. I joined a team using J2EE version 1.3 to build a distributed gift card processor. The point-of-sale system was written in Microsoft Visual C++ 6.0. We were just learning about this new thing called SOAP, the Simple Object Access Protocol. The running joke was that it was too ill defined to be called a protocol, that it was not about accessing objects, and it was anything but simple. But it did hold some promise for making a C++ client talk to a Java server.

We all added three new books to our libraries. The first was on implementing a SOAP client in C++. The second was on JAXP, the Java API for XML Processing. And the third detailed the operation and limitations of TCP/IP. Armed with these tools, we began to build.

At first, the challenge was just to get the two platforms to talk to each other. When we finally settled on a subset of SOAP that both sides could handle, we thought we were over the hump. Little did we know that on the other side lay mountains.

There were reliability problems with the network. We set up a lab that continually ran transactions every night. We would check the card balances in the morning to find that some machines would have the wrong total. That led to a day of digging through logs, setting up the next test run, and then leaving it going until morning.

Over time, we evolved a message exchange protocol (over SOAP) based on confirmations and acknowledgments. One side sent a message. The next morning, we found messages missing. So next, the recipient confirmed that the message arrived. The next morning, we found duplicates. And so the sender acknowledged the confirmation. Fewer missing messages, but still not perfect.

It took many failed releases and many years of busy holiday seasons to work through all of the problems. We learned about the Two Generals' Problem (TGP) and realized why our message exchange protocol was flawed. Then we learned about eventual consistency and designed a working solution. This solution required that there be some uncertainty about how much money was left on a gift card. We tried to have that conversation with the product owner. Bankers get eventual consistency of money. Our product owner was not a banker.

The lessons we learned from gift cards were learned the hard way. "Guaranteed delivery" does not mean what you think it means. You need to first move data, then process it. Remote procedure calls (RPCs) aren't procedure calls. There is no line of code in a client-server system before which the transaction rolls back and after which it commits. I didn't want to learn those lessons over and over again.

And so I started putting those lessons together and defining a system that I called historical modeling. It was based on the idea that historical facts cannot be modified or destroyed. It relied upon the predecessor/successor relationships among facts. And it identified facts based only on their content, not on their location. I filled a notebook with examples of historical models. Eventually, I gained an intuitive feel for which kinds of solutions could be modeled historically and which could not. That's when I knew that I had to share it. Hopefully I could save someone else the pain of learning these lessons the hard way.

Since then, I have had countless conversations about immutable architectures. I broke the topic down into digestible chunks for conference and user group talks. I produced online courses that taught idempotent and commutative messaging. Yet none of that has truly empowered others to begin practicing immutability themselves. It can't just be adopted in pieces. Taking on only a subset of the ideas leaves gaps that can only be filled with the rest of the system.

Finally, I packaged the entire system in two forms. One, the open source project Jinaga. And two, the book that you are now holding. This is a complete treatment of the system, the patterns, and the techniques. It anticipates the problems that historical modeling creates and provides the solutions that enable a cohesive implementation. Most importantly, it presents the mathematical foundation that makes the technique work.

If you have read this far into the introduction, you have probably faced some of these same problems. You might even have come up with similar solutions. This leaves only a few more questions you probably have about this book. Who should read it? What will I get out of it? How is it organized? And how do I go about reading it?

Glad you asked.

Who Should Read It

This book is intended primarily for three audiences: decision makers, system builders, and tool crafters. You are a decision maker if you identify the problems for which you want to create solutions. Your title might be CTO, product owner, or business systems

analyst. There are some problems that you can outsource, some that you can buy solutions for, and some that define your core business value. You need to find just the right team to build solutions to problems of this third kind. To find them, you need to be able to talk to them. And once you've brought them on board, you need to understand what they are doing. If your core business problem looks like the kind of thing that can be solved with an immutable architecture, this book will help you build that team and have those conversations.

Or perhaps you are a system builder. You are a member of the team brought in to deliver value against a core business domain. Your title might be developer, test engineer, or user experience designer. You know how to solve problems. But it would be great to have some ready-made solutions to the most common problems of distributed computing. You want to know that all of the edge cases are accounted for. You desire a common language to talk about solutions with the people who are helping you build them. If your software development challenges require constructing eventually consistent distributed systems, then this book will give you those tools.

Finally, you – like me – might be a tool crafter. You are a force multiplier. The things that you build empower others to build solutions more quickly, more predictably, and more effectively. You might be a solutions architect or an open source maintainer. If you have a team, you want them focused on delivering business value while you take care of the plumbing. If you serve the community, you want consumers to be able to quickly learn and apply your framework to build robust systems. In either case, this book lays out the mathematics, algorithms, and patterns that assure the correctness of your solutions.

What You Will Get Out of It

I have a secret. This is a math book. Don't tell anybody who hasn't read this far into the introduction.

Mathematics is the greatest invention of humankind. It is surprising in its ability to describe the natural world. It is astonishingly applicable to a broad range of problems. And it is the only way that we can be sure of anything.

The way that we normally learn that we have gotten something right is to test it. We'll put our solution in one situation and see if we get the expected result. Then we'll try another scenario and see what it does. If we are really good, then we can imagine a few unexpected conditions and test for those. But the unexpected is really hard to anticipate.

Testing is all about gathering empirical evidence. It only gives you confidence that the system behaves as expected in certain cases. It does not give you any assurance that you haven't missed something.

Knowing requires mathematical deduction. If something is proven mathematically, then you can be sure that it will be true no matter what test case you try. Pythagoras is true for any right triangle. Euclid holds up for all figures on the plane. If your reasoning is sound, you can be sure that you haven't missed any edge cases.

It's not that mathematical truths are universal. It's that they come with known limitations. Division only works for nonzero divisors. Pythagoras only holds on the plane. The rules of deduction tell us how to carry those boundaries through to the solution so that we know precisely where that solution applies and where it doesn't.

This book applies mathematical rigor to the problem of distributed computing. It is not the first to do so, but it does provide a complete and practical solution. If you follow the deductive reasoning over the problem and carry the limitations of distributed systems through your calculations, you will end up with an understanding of the boundaries of the solution. This book is your guide through that process.

How It Is Organized

The book is roughly divided into three parts, analogous to the three primary audiences. Decision makers need only read the first part, which includes the first three chapters. In this part, you first learn why immutability is so important. Then you explore the space of alternatives, eventually landing on historical modeling. Finally, you learn how to read a historical model so that you can communicate more effectively with your team. You can stop reading when we get into some deep math.

System builders will want to continue on to the second part. This includes Chapters 4 through 9. First, we see how to apply immutability to analyzing systems. Then, we get neck deep in the mathematical foundations of immutability, causality, and conflict-free replicated data types (CRDTs). Next, we learn how system operators will compose solutions from these components. And finally, we study patterns for modeling entities, building state machines, and enforcing security rules. These are the tools that you will need to build robust distributed systems.

My people, the tool crafters, will want to read right through to the end. We'll start with techniques for using traditional technologies like relational databases, REST APIs, and message queues. This will help prepare you for a gradual transition from

stateful to immutable architectures. After that, we'll see how to construct libraries and infrastructure components purpose built for immutability. We pull it all together and describe an ecosystem made up of collaborative applications generating emergent behavior from shared specifications. That's where we get into the mathematical results that I find truly beautiful and inspiring. I hope you follow me to the end.

How to Read It

Now that you know this is a math book, you might have some reservations about how you are going to read it. Perhaps you struggled through algebra or dropped out of calculus. You might think that math is not for you.

It is my belief that math is for everyone. And it is my goal with this book to prove it. Mathematics is nothing more than applying logical reasoning over symbolic representations of abstract concepts. Programming, on the other hand, is applying logical operations to a symbolic language describing generic rules. In other words, they are the same thing. If you are a programmer, then you are an applied mathematician.

One problem with mathematics is the jargon. In order to efficiently communicate with each other, mathematicians have to come up with words to represent ideas. Unfortunately, natural language is limited, and all of the good words are taken. And so mathematicians either make up new words or use terms that almost mean the right thing. One example is the term "join semilattice." How does the structure of a rose trellis relate to eventual consistency? In this book, I don't use that term even though I talk about that concept. And where I can't avoid jargon, I will clearly define the terms.

Another problem with mathematics is how it is written. Math papers have a predictable form. They start with an abstract. Then they fully define the problem. What follows is section after section of lemmas and propositions building an argument. Every statement is justified by the statements before, until finally, like an M. Night Shyamalan plot twist, one final assertion puts the whole argument into perspective and the result emerges.

While I really enjoy a good math paper, I don't read them the way that they are written. I skim the first few paragraphs for the motivation behind the problem. I scan the headings for the outline of the argument. I want to know why each statement is proven and how it will contribute to the whole. I want to know how the story is going to play out before I invest the time in understanding it.

I wrote this book the way that I read a math paper. In each section, you will understand the motivation behind a certain result. Then you will see a sketch of the basic reasoning. There will be no mystery why each of the steps is there. Then the section will justify each of those steps with the rigor they require.

I fully anticipate that this will impact the way you read the book. If you are after results, you can read just a paragraph or two past the section header. If you want to know why or how, then continue a bit further to understand the argument. And if you need to be convinced, then finish out the whole section. The important thing is that you can stop reading whenever it gets too deep and skip to the next section. You won't miss anything important to you.

If you have read this section without skipping anything, then I am truly pleased to have you. You are one of my people. With your help, we can build the software that the world needs. We will make it reliable, efficient, and correct. And it will give our users the autonomy they need to do their jobs with creativity and confidence, knowing that we have provided the mathematical rigor.

PART I

Definition

CHAPTER 1

Why Immutable Architecture

Distributed systems are hard.

Most of us have used a website to buy a product. You might have seen a purchase page that contains a warning **do not click submit twice!** Maybe you've used a site that simply disables the buy button after you click it. The authors of that site have run up against one of the hard problems of distributed systems and did not know how to solve it. They abdicated the responsibility of preventing duplicate charges to the consumer.

Maybe you've used a mobile application on a train. The train enters a tunnel just as you save some data. The mobile app spins for a few seconds before you realize that you are in a race. Will the train leave the tunnel before the app gives up? Will the app correct itself once the connection is reestablished? Or will you lose your data and have to enter it again?

If you are involved in the creation of distributed systems, you are expected to find, fix, and prevent these kinds of bugs. If you are in QA, it is your job to imagine all of the possible scenarios and then replicate them in the lab. If you are in development, you need to code for all of the various exceptions and race conditions. And if you are in architecture, you are responsible for cutting the Gordian Knot of possible failures and mitigations. This is the fragile process by which we build the systems that run our society.

The Immutability Solution

Distributed systems are hard to write, test, and maintain. They are unreliable, unpredictable, and insecure. The process by which we build them is certain to miss defects that will adversely affect our users. But it is not your fault. As long as we depend upon individuals to find, fix, and mitigate these problems, defects will be missed.

M. L. Perry, *The Art of Immutable Architecture*, https://doi.org/10.1007/979-8-8688-0288-1_1

This book explores a different process for building distributed systems. Rather than connecting programs together and testing away the defects, this approach starts with a fundamental representation of the business problem that *spans* machines. And this fundamental representation is *immutable*.

On its face, immutability is a simple concept. Write down some data, and ensure that it never changes. It can never be modified, updated, or deleted. It is indelible. Immutability solves the problem of distributed systems for one simple reason: every copy of an immutable object is just as good as any other copy. As long as things never change, keeping distant copies in sync is a trivial problem.

The Problems with Immutability

Unfortunately, immutability is counter to how computers actually work. A machine has a limited amount of memory. Machines work by modifying the contents of memory locations over time to update their internal state. So the first problem of modeling immutable data on a computer is how to represent it in fixed mutable memory.

The second problem is that when we look out at the world of problems that we want to solve, we see change. People change their names, addresses, and phone numbers. Bank account balances go up and down. Property changes hands and ownership is transferred. How then are we to model a changing problem space with unchanging data?

Our initial instinct is to model the mutable world within the mutable space of the computer. This is the solution that has led us to build programs and databases based on mutation. Programs have assignment statements; databases have UPDATE statements. When we connect those programs and databases together to create distributed systems, crazy unpredictable behaviors emerge. And we are left with the unending task of testing until all of those anomalies are gone.

Redefine the Process

This book defines a new process by which to build distributed systems. It relies upon a rigorous system of specification, a mathematical proof of correctness, and a mechanical translation into machine behavior.

The first step is to model the business domain as one large immutable data structure. We call this data structure a *historical model*. The goal is not for a single machine or database to house the entire structure. It is instead to share that structure across nodes. The historical model is a description that both humans and machines can understand and reason about, not a concrete implementation.

The second step is to subdivide that model into autonomous components. This subdivision will not be clean; there will be overlap. We will use that overlap to derive the rules, messages, and protocols by which machines communicate with one another.

The third step is to convert these subdivisions into deployable software. This step is mechanical: a machine can do it. We call the system that supports this process an *immutable runtime*. One such runtime – Jinaga – is currently in operation and will serve as the reference implementation. For organizations not yet ready to adopt an immutable runtime, this book describes how to perform this step manually. You can build autonomous components from traditional databases, protocols, and messaging infrastructure. Be careful, however. Without the mechanisms of the immutable runtime, you will still be prone to the errors of human implementation.

This system is based on prior art, most notably conflict-free replicated data types (CRDTs). Throughout this book, we will reference that research in the form of math and computer science papers. Every claim is justified. I humbly add two new claims to this body of work. Both are based on projections — the ways in which you extract information from a replica. The first claim is that replicas will reach consistency after exchanging a subset of updates determined by a set of projections. And the second is that we can determine which projections produce new results after receiving an update. These two claims allow us to automate message passing and cache invalidation in ways that are impossible without the assumption of immutability. The proofs of these claims constitute the last two chapters of the book.

It is my ambition that you build a historical model of your own business domain. From this, you will construct more reliable, resilient, and secure distributed systems, whether using an immutable runtime or by hand. Let's begin by understanding the problem of distributed computing.

The Fallacies of Distributed Computing

Between 1991 and 1997, engineers at Sun Microsystems collected a list of mistakes that programmers commonly make when writing software for networked computers. Bill Joy, Dave Lyon, L Peter Deutsch, and James Gosling cataloged eight assumptions that developers commonly hold about distributed computing. These assumptions, while obviously incorrect when stated explicitly, nevertheless inform many of the decisions that the Sun engineers found in systems of the day.

The fallacies are these:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Although it has been years since that list was written, many of these assumptions continue to be common. I can recall on several occasions being surprised that a program that worked flawlessly on *localhost* failed quickly when deployed to a test environment. The program contained hidden assumptions that the network was reliable, that latency was zero, and that the topology doesn't change. Here are examples of just these three.

The Network Is Not Reliable

One way in which these fallacies appear in modern systems is when a remote API is presented as if it were a function call. Several platform services have promoted this abstraction, including remote procedure calls, .NET Remoting, Java Remote Method Invocation, Distributed COM, SOAP, and SignalR. When a remote invocation is made to look like a local function call, it is easy for a developer to forget that the network is not reliable.

Any time you call a function, you can rest assured that execution will continue with its first line. And if the function makes it to the return statement, you can feel pretty confident that the next line to run will be the one following the function call. Remote procedure calls, however, make no such claims. They can fail on invocation or on return. The calling code will be unable to tell which.

An abstraction that hides the fact of a network hop does a disservice to its consumers. In an effort to make things easier and more familiar, it pretends that an inconvenient truth can be ignored. Such abstractions make it easier for developers to believe the fallacy that the network is somehow reliable.

Latency Is Not Zero

Modern web applications have moved away from the client proxy in favor of more explicit REST APIs. These APIs avoid the mistake of presenting the remote machine as if it were a library of functions that could be invoked reliably. They instead present the world as a web of interconnected resources, each responding to a small set of HTTP verbs. Unfortunately, this style of programming makes it easy to forget that latency is not zero.

Some of the HTTP verbs are guaranteed to be *idempotent*. If the client duplicates the request, the server promises not to duplicate the effect. There is no way for the protocol to enforce that guarantee, but server-side applications typically uphold the contract. Examples of HTTP verbs that are idempotent are `PUT` and `PATCH`. An HTTP verb that is *not* guaranteed to be idempotent is `POST`.

On the Web, HTTP `POST` is often used to submit a form. When a web application responds quickly, the lack of idempotency guarantee makes little difference. But as latency increases, the user starts to wonder if they actually clicked the submit button. And if that button triggered a purchase, they have to wonder if they will be charged twice if they try again. An end user has no good recourse during an extended latency after clicking a *Buy* button, nor does a client-side application developer have a good response to a timeout on `POST`.

There is no correct use of an API that features non-idempotent network requests. Because latency is not zero, there will always be a time during which the client is unsure if the server has received the request. As latency exceeds the time that the client is willing to wait, they must make a choice: either abort the attempt or retry. If the client aborts, then they don't know whether the request has been processed. And if they retry, then the effect might be duplicated.