Type-Aware Programming Models for Distributed Applications

Brandon Holt

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Luis Ceze, Chair

Mark Oskin, Chair

Dan Ports

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Type-Aware Programming Models for Distributed Applications

Brandon Holt

Co-Chairs of the Supervisory Committee:
Associate Professor Luis Ceze
UW CSE

Associate Professor Mark Oskin
UW CSE

[write me!]

Contents

1. Introduction

Modern applications have grown beyond what a single machine is capable of handling. It is now hard to even think of compelling new applications that require no external compute, cloud storage, or communication among users. From social networking sites and games (Facebook, Pokemon Go), to online retail services (Amazon, Etsy) and collaborative working tools (Github, Google Docs) — all of these are split between client devices and servers distributed among many machines spread among datacenters around the world.

Building software that spans these varied machines is fundamentally more challenging than traditional standalone applications. Consider the architecture of a modern web service such as Twitter: the mobile phone app communicates over the wide-area internet with frontend servers in one of Twitter's datacenters. This frontline of servers, however, merely routes requests to some other set of services which gather data for the user, such as their timeline of recent tweets, recommendations for users to follow, and advertisements. Each of these in turn is itself a distributed application, running across multiple machines, with data stored in each machine's memory, in other caching services, and in slower persistent storage. Designing this kind of software requires decisions about the protocols and APIs each component uses to communicate with the others, where each piece of functionality should reside, as well as countless other questions about storage systems to use and low-level implementation details.

In addition to purely functional design choices, distributed applications introduce many fundamentally difficult performance decisions, such as

- **Replication and consistency:** Data is often replicated for fault tolerance and high availability, but this creates a tension between availability (e.g. quick responses in the face of failure) and consistency (correct

values that everyone agrees on).

- **Sharding and locality:** Not all data will fit on a single machine, so it must be split among many machines. But where should data be placed? What should go together? How much data must be moved to compute something?

- **Parallelism and synchronization:** More machines means more compute resources but requires more coordination to ensure correct results. At what point does the additional synchronization outweigh the benefit of parallelism? Which ordering constraints are actually necessary? Where are the true serialization bottlenecks?

Developing traditional standalone programs, we are used to a suite of tools that support building correct and efficient software. Most notably, we rely on programming languages and compilers to understand the programs we write and optimize them for the machine they run on. Type systems prevent common mistakes like assigning an incorrect value to a variable or dereferencing a null pointer. Compilers and runtime systems support programmers by freeing them from burdens such as explicit memory management. Finally, hardware automatically manages data, caching it close the cores that are likely to use it again, moving data around transparently to wherever it is needed.

When it comes to distributed applications, however, developers are on the hook. They must ensure they use APIs correctly, choose the correct level of consistency, and distribute their data among services explicitly. Most of the knowledge about application semantics is lost at the boundary between each service; only information that is explicitly shared through the provided interface can be leveraged by the system. Take a typical application using a key-value store for its persistent state. The application may have a rich hierarchy of classes and data structures that represent its data. However, if the key-value store's interface only provides `put` and `get` operations on byte strings, then most of this structure will be lost when the application sends its data to the storage system. Accordingly, the key-value store has no chance of optimizing for the particular use case of this application. It can't predict which data is more likely to be accessed together, or understand when two updates to the

same key could be performed independently, or know which operations are the most important to get right. However, if the interface is expressive enough to provide more of these semantics from the application and the system is designed to take advantage of that additional knowledge to inform how it handles that data, then programmers can benefit from optimizations across the many dimensions of performance.

1.1. Overview

The overarching goal of this dissertation is to help programmers express their distributed application's needs to the system, and then design and build systems that can take advantage of that knowledge to improve performance. This has been borne out in several distinct projects which leverage different pieces of knowledge to optimize for different situations.

1.1.1. Alembic

A compiler that optimizes for locality within distributed applications, choosing when to move computations closer to data using knowledge of remote data accesses expressed through new "global" pointers in C++, using the distributed shared memory runtime system we developed called Grappa.

1.1.2. Contention-avoiding Distributed Data Structures

Designing globally shared data structures for our distributed runtime system, Grappa, that efficiently handle the contention resulting from the massive number of threads which concurrently update them.

1.1.3. Claret

Leveraging the semantics of abstract data types that represent application state to improve the performance of distributed transactions, using properties such as commutativity and associativity to reduce synchronization and conflicts.

1.1.4. Disciplined Inconsistency

Helping programmers safely trade off consistency for performance by allowing them to express performance and correctness criteria in applications using replicated datastores and using these constraints to dynamically adjust consistency within the datastore while statically ensuring consistency safety.

1.2. Themes

The projects above contribute to the ever-growing, though still impoverished, toolkit available to developers of distributed applications. This document is organized according to these discrete projects, but several common threads (pardon the parallelism pun) tie the work together. Distributed applications face the same few fundamental challenges that permeate this work. Luckily, this is not strictly a zero-sum game; by using additional knowledge the programmer may already have, many of these challenges can be mitigated. The trick lies in exposing situations where prior solutions had to be overly conservative due to lack of insight. We will see many instances of these themes crop up throughout later chapters, but here we give you a hint of what to watch for.

1.2.1. Locality and synchronization

Data layout has always been important for application performance. However, distributed applications must be particularly concerned about where their data resides because retrieving data from remote machines can be orders of magnitude more expensive than main memory. This cost comes partly from simple latency resulting from physical distance, but software overhead within the complex network stack and operating system interactions make for significant cost even among neighboring machines.

Locality also plays a role in synchronization. In general, guaranteeing linearizability — where all observers agree on a single total order of operations on an object or record — requires sequencing concurrent operations. This is typically done by designating one owner (such as a single thread) which orders all operations that it receives. Note that this

is analogous to how coherence protocols work in multi-processors. As an aside, multiple machines can coordinate to agree on a common ordering using expensive consensus protocols (e.g. Paxos), but this is done for fault tolerance, not performance, and typically comes down to choosing a leader which can trivially order operations.

Because of this reliance on a single owner to order operations, applications must choose carefully where to place data and computation. Heavily modified data cannot easily be cached, but with the high cost of communication, it is important to ensure that computation happens as close as possible to the data it uses. Chapter 2 *(Alembic)* proposes a compiler that can leverage its knowledge of data accesses to move computation (essentially migrating threads) closer to their data, even if it is spread among multiple machines. Both Chapter 3 and Chapter 4 use the idea of *combining* to get around the single-owner problem by pre-synchronizing operations in parallel before sequencing them.

1.2.2. Consistency and ordering

Consistency models[1] allow programmers to reason about the behavior of reads and writes to replicated data, particularly properties specifying the observable order of updates. The weakest, eventual consistency, merely guarantees that at some point in the future, all replicas will agree, but says nothing about when or which updates may be reflected. On the other hand, *read your writes* guarantees, as the name implies, that reads will reflect at least the last write made by the same client — a primitive guarantee compared with total orderings, but sufficient for some use cases.

The problem with consistency models is that they imply an ordering between operations that almost certainly does not reflect what a particular application needs or wants. Sequential consistency, which forces all operations to be totally ordered, imposes extreme restrictions on reorderings and precludes high availability, yet eventual consistency is so

[1]The term *consistency* means something different to everyone, from the "C" in "ACID", to the "C" in "CAP", not to mention to computer architects who believe in "sequential consistency". In this work, we use it to discuss the behavior of replicated data, e.g. *eventual consistency*.

permissive as to give users nearly no idea what their reads could be. [far too strong, and where is this going?]

1.2.3. Abstract data types

2. Alembic

3. Combining

4. Claret

Bibliography

[1] Apache Software Foundation. Cassandra. http://cassandra.apache.org/, 2015.

[2] Basho Technologies, Inc. Riak. http://docs.basho.com/riak/latest/, 2015.

[3] Brady Forrest. Bing and google agree: Slow pages lose users. Radar, June 2009. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

[4] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[5] Greg Linden. Make data useful. Talk, November 2006. http://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html.

[6] Cade Metz. How instagram solved its justin bieber problem, November 2015. URL http://www.wired.com/2015/11/how-instagram-solved-its-justin-bieber-problem/.

[7] Dao Nguyen. What it's like to work on buzzfeed's tech team during record traffic. http://www.buzzfeed.com/daozers/what-its-like-to-work-on-buzzfeeds-tech-team-during-record-t, February 2015.

[8] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles - SOSP'11*, SOSP. Association for Computing Machinery (ACM), 2011. doi:10.1145/2043556.2043592.

[9] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'09, pages 43–58, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558981.

[10] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13*. ACM Press, 2013. doi:10.1145/2517349.2522731.