White Papers You Must Read as a Senior Engineer (You'll Thank Me Later):

MapReduce

Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

Simplified data processing on large clusters

TAO

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov
Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov
Dmitri Petrov, Lovro Puzar, Yee Jiun Song, Venkat Venkataramani
Facebook, Inc.

Abstract

We introduce a simple data model and API tailored for serving the social graph, and TAO, an implementation of this model. TAO is a geographically distributed data store that provides efficient and timely access to the social graph for Facebook's demanding workload using a fixed set of queries. It is deployed at Facebook, replacing memcache for many data types that fit its model. The system runs on thousands of machines, is widely distributed, and provides access to many petabytes of data. TAO can process a billion reads and millions of writes each second.

1 Introduction

Facebook has more than a billion active users who record their relationships, share their interests, upload text, images, and video, and curate semantic information about their data [2]. The personalized experience of social applications comes from timely, efficient, and scalable access to this flood of data, the *social graph*. In this paper we introduce TAO, a read-optimized graph data store we have built to handle a demanding Facebook workload.

Before TAO, Facebook's web servers directly accessed MySQL to read or write the social graph, aggressively using memcache [21] as a lookaside cache. TAO implements a graph abstraction directly, allowing it to avoid some of the fundamental shortcomings of a lookaside cache architecture. TAO continues to use MySQL for persistent storage, but mediates access to the database and uses its own graph-aware cache.

TAO is deployed at Facebook as a single geographically distributed instance. It has a minimal API and explicitly favors availability and per-machine efficiency over strong consistency; its novelty is its scale: TAO can sustain a billion reads per second on a changing data set of many petabytes.

Overall, this paper makes three contributions. We motivate (§ 2) and characterize (§ 7) a challenging workload: efficient and available read-mostly access to a changing graph. We describe objects and associations, a data model and API that we use to access the graph (§ 3). Lastly, we detail TAO, a geographically distributed system that implements this API (§§ 4–6), and evaluate its performance on our workload (§ 8).

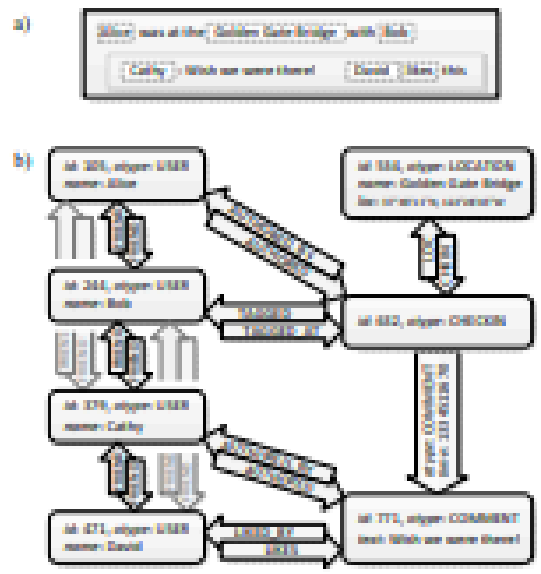


Figure 1: A running example of how a user's checkin might be mapped to objects and associations.

2 Background

A single Facebook page may aggregate and filter hundreds of items from the social graph. We present each user with content tailored to them, and we filter every item with privacy checks that take into account the current viewer. This extreme customization makes it infeasible to perform most aggregation and filtering when content is created; instead we resolve data dependencies and check privacy each time the content is viewed. As much as possible we pull the social graph, rather than pushing it. This implementation strategy places extreme read demands on the graph data store; it must be efficient, highly available, and scale to high query rates.

2.1 Serving the Graph from Memcache

Facebook was originally built by storing the social graph in MySQL, querying it from PHP, and caching results in memcache [21]. This lookaside cache architecture is well suited to Facebook's rapid iteration cycles, since all

Facebook's distributed data store for the social graph

Bitcoin

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buyers. In this paper, we propose a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

A Peer-to-Peer electronic cash system

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {sanjay,hgobioff,shuntak}@google.com.

SOSP'03, October 19-22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

A Scalable distributed file system

Bigtable

Fay Chang, Jeffrey Dean, Sanjay Ghemawat,
Wilson C. Hsieh, Deborah A. Wallach Mike Burrows,
Tushar Chandra, Andrew Fikes, Robert E. Gruber

fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

To appear in OSDI 2006

A Distributed storage system for structured data

CAP Theorem

Seth Gilbert
National University of Singapore

Nancy A. Lynch
Massachusetts Institute of Technology

Abstract

Almost twelve years ago, in 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between *consistency*, *availability*, and *partition tolerance*. This trade-off, which has become known as the *CAP Theorem*, has been widely discussed ever since. In this paper, we review the CAP Theorem and situate it within the broader context of distributed computing theory. We then discuss the practical implications of the CAP Theorem, and explore some general techniques for coping with the inherent trade-offs that it implies.

1 Introduction

Almost twelve years ago, in 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between *consistency*, *availability*, and *partition tolerance*. This trade-off, which has become known as the *CAP Theorem*, has been widely discussed ever since.

Theoretical context. Our first goal in this paper is to situate the CAP Theorem in the broader context of distributed computing theory. Some of the interest in the CAP Theorem, perhaps, derives from the fact that it illustrates a more general trade-off that appears everywhere (e.g., [4,7,15,17,23]) in the study of distributed computing: the impossibility of guaranteeing both *safety* and *liveness* in an *unreliable distributed system*:

Safety: Informally, an algorithm is *safe* if nothing bad ever happens. A quiet, uneventful room is perfectly safe. Consistency (as defined in the CAP Theorem) is a classic safety property: every response sent to a client is correct.

Liveness: By contrast, an algorithm is *live* if eventually something good happens. In a busy pub, there may be some good things happening, and there may be some bad things happening—but overall, it is quite lively. Availability is a classic liveness property: eventually, every request receives a response.

Unreliable: There are many different ways in which a system can be unreliable. There may be partitions, as is discussed in the CAP Theorem. Alternatively, there may be crash failures, message loss, malicious attacks (or Byzantine failures), etc.

The CAP Theorem, in this light, is simply one example of the fundamental fact that you cannot achieve both safety and liveness in an unreliable distributed system.

Practical implications. Our second goal in this paper is to discuss some of the practical implications of the CAP Theorem. Since it is impossible to achieve both consistency and availability in an unreliable system, it is necessary in practice to sacrifice one of these two desired properties. On the one hand, there are systems that guarantee strong consistency and provide *best effort* availability. On the other hand, there are systems that guarantee availability, and provide *best effort* consistency. Perhaps surprisingly, there is a third option: some systems may sacrifice both consistency and availability! In doing so they may achieve a trade-off better suited for the application at hand.

Roadmap. We begin in Section 2 by reviewing the CAP Theorem, as it was formalized in [16]. We then examine in Section 3 how the CAP Theorem fits into the general framework of a trade-off between safety and liveness. Finally, in Section 4, we discuss the implications of this trade-off, and various strategies for coping with it. We conclude in Section 5 with a short discussion of some new directions to consider in the context of the CAP Theorem.

Perspectives on the CAP theorem

Kafka

Jay Kreps
LinkedIn Corp.
jkreps@linkedin.com

Neha Narkhede
LinkedIn Corp.
nnarkhede@linkedin.com

Jun Rao
LinkedIn Corp.
jrao@linkedin.com

ABSTRACT

Log processing has become a critical component of the data pipeline for consumer internet companies. We introduce Kafka, a distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency. Our system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. We made quite a few unconventional yet practical design choices in Kafka to make our system efficient and scalable. Our experimental results show that Kafka has superior performance when compared to two popular messaging systems. We have been using Kafka in production for some time and it is processing hundreds of gigabytes of new data each day.

General Terms

Management, Performance, Design, Experimentation.

Keywords

messaging, distributed, log processing, throughput, online.

1. Introduction

There is a large amount of "log" data generated at any sizable internet company. This data typically includes (1) user activity events corresponding to logins, pageviews, clicks, "likes", sharing, comments, and search queries; (2) operational metrics such as service call stack, call latency, errors, and system metrics such as CPU, memory, network, or disk utilization on each machine. Log data has long been a component of analytics used to track user engagement, system utilization, and other metrics. However recent trends in internet applications have made activity data a part of the production data pipeline used directly in site features. These uses include (1) search relevance, (2) recommendations which may be driven by item popularity or co-occurrence in the activity stream, (3) ad targeting and reporting, and (4) security applications that protect against abusive behaviors such as spam or unauthorized data scraping, and (5) newsfeed features that aggregate user status updates or actions for their "friends" or "connections" to read.

This production, real-time usage of log data creates new challenges for data systems because its volume is orders of magnitude larger than the "real" data. For example, search, recommendations, and advertising often require computing granular click-through rates, which generate log records not only for every user click, but also for dozens of items on each page that are not clicked. Every day, China Mobile collects 5–8TB of phone call records [11] and Facebook gathers almost 6TB of various user activity events [12].

Many early systems for processing this kind of data relied on physically scraping log files off production servers for analysis. In recent years, several specialized distributed log aggregators have been built, including Facebook's Scribe [6], Yahoo!'s Data Highway [4], and Cloudera's Flume [3]. These systems are primarily designed for collecting and loading the log data into a data warehouse or Hadoop [8] for offline consumption. At LinkedIn (a social network site), we found that in addition to traditional offline analytics, we needed to support most of the real-time applications mentioned above with delays of no more than a few seconds.

We have built a novel messaging system for log processing called Kafka [18] that combines the benefits of traditional log aggregators and messaging systems. On the one hand, Kafka is distributed and scalable, and offers high throughput. On the other hand, Kafka provides an API similar to a messaging system and allows applications to consume log events in real time. Kafka has been open sourced and used successfully in production at LinkedIn for more than 6 months. It greatly simplifies our infrastructure, since we can exploit a single piece of software for both online and offline consumption of the log data of all types. The rest of the paper is organized as follows. We revisit traditional messaging systems and log aggregators in Section 2. In Section 3, we describe the architecture of Kafka and its key design principles. We describe our deployment of Kafka at LinkedIn in Section 4 and the performance results of Kafka in Section 5. We discuss future work and conclude in Section 6.

2. Related Work

Traditional enterprise messaging systems [1][7][15][17] have existed for a long time and often play a critical role as an event bus for processing asynchronous data flows. However, there are a few reasons why they tend not to be a good fit for log processing. First, there is a mismatch in features offered by enterprise systems. These systems often focus on offering a rich set of delivery guarantees. For example, IBM Websphere MQ [7] has transactional supports that allow an application to insert messages into multiple queues atomically. The JMS [14] specification allows each individual message to be acknowledged after consumption, potentially out of order. Such delivery guarantees are often overkill for collecting log data. For instance, losing a few pageview events occasionally is certainly not the end of the world. These unneeded features tend to increase the complexity of both the API and the underlying implementation of these systems. Second, many systems do not focus as strongly on throughput as their primary design constraint. For example, JMS has no API to allow the producer to explicitly batch multiple messages into a

A Distributed messaging system for log processing

The Chubby

Mike Burrows, *Google Inc.*

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

1 Introduction

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example, a Chubby instance (also known as a Chubby cell) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. Most Chubby cells are confined to a single data centre or machine room, though we do run at least one Chubby cell whose replicas are separated by thousands of kilometres.

The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals included reliability, availability to a moderately large set of clients, and easy-to-understand semantics; throughput and storage capacity were considered secondary. Chubby's client interface is similar to that of a simple file system that performs *whole-file* reads and writes, augmented with advisory locks and with notification of various events such as file modification.

We expected Chubby to help developers deal with coarse-grained synchronization within their systems, and in particular to deal with the problem of electing a leader from among a set of otherwise equivalent servers. For example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the *distributed consensus* problem, and realize we require a solution using *asynchronous* communication; this term describes the behaviour of the vast majority of real networks, such as Ethernet or the Internet, which allow packets to be lost, delayed, and reordered. (Practitioners should normally beware of protocols based on models that make stronger assumptions on the environment.) Asynchronous consensus is solved by the *Paxos* protocol [12, 13]. The same protocol was used by Oki and Liskov (see their paper on viewstamped replication [19, §4]), an equivalence noted by others [14, §6]. Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core. Paxos maintains safety without timing assumptions, but clocks must be introduced to ensure liveness; this overcomes the impossibility result of Fischer *et al.* [5, §1].

Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it. In the sections that follow, we describe Chubby's design and implementation, and how it

The lock service for loosely coupled distributed systems

The Log Structured Merge Tree

Patrick O'Neil[1], Edward Cheng[2]
Dieter Gawlick[3], Elizabeth O'Neil[1]
To be published: Acta Informatica

ABSTRACT. High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the History for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve cost-performance in domains where disk arm costs for inserts with traditional access methods overwhelm storage media costs. The LSM-tree approach also generalizes to operations other than insert and delete. However, indexed finds requiring immediate response will lose I/O efficiency in some cases, so the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for History tables and log files, for example. The conclusions of Section 6 compare the hybrid use of memory and disk components in the LSM-tree access method with the commonly understood advantage of the hybrid method to buffer disk pages in memory.

1. Introduction

As long-lived transactions in activity flow management systems become commercially available ([10], [11], [12], [20], [24], [27]), there will be increased need to provide indexed access to transactional log records. Traditionally, transactional logging has focused on aborts and recovery, and has required the system to refer back to a relatively short-term history in normal processing with occasional transaction rollback, while recovery was performed using batched sequential reads. However, as systems take on responsibility for more complex activities, the duration and number of events that make up a single long-lived activity will increase to a point where there is sometimes a need to review past transactional steps in real time to remind users of what has been accomplished. At the same time, the total number of active events known to a system will increase to the point where memory-resident data structures now used to keep track of active logs are no longer feasible, notwithstanding the continuing decrease in memory cost to be expected. The need to answer queries about a vast number of past activity logs implies that indexed log access will become more and more important.

[1]Dept. of Math & C.S, UMass/Boston, Boston, MA 02125-3393, {poneil | eoneil}@cs.umb.edu
[2]Digital Equipment Corporation, Palo Alto, CA 94301, edwardc@pa.dec.com
[3]Oracle Corporation, Redwood Shores, CA, dgawlick@us.oracle.com

Data structure for efficiently storing key-value pairs in storage systems

Spanner

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

Simplified data processing on large clusters

Consistent Hashing and Random Trees

David Karger[1] Eric Lehman[1] Tom Leighton[1,2] Matthew Levine[1] Daniel Lewin[1]
Rina Panigrahy[1]

Abstract

We describe a family of caching protocols for distributed networks that can be used to decrease or eliminate the occurrence of hot spots in the network. Our protocols are particularly designed for use with very large networks such as the Internet, where delays caused by hot spots can be severe, and where it is not feasible for every server to have complete information about the current state of the entire network. The protocols are easy to implement using existing network protocols such as TCP/IP, and require very little overhead. The protocols work with local control, make efficient use of existing resources, and scale gracefully as the network grows.

Our caching protocols are based on a special kind of hashing that we call *consistent hashing*. Roughly speaking, a consistent hash function is one which changes minimally as the range of the function changes. Through the development of good consistent hash functions, we are able to develop caching protocols which do not require users to have a current or even consistent view of the network. We believe that consistent hash functions may eventually prove to be useful in other applications such as distributed name servers and/or quorum systems.

1 Introduction

In this paper, we describe caching protocols for distributed networks that can be used to decrease or eliminate the occurrences of "hot spots". Hot spots occur any time a large number of clients wish to simultaneously access data from a single server. If the site is not provisioned to deal with all of these clients simultaneously, service may be degraded or lost.

Many of us have experienced the hot spot phenomenon in the context of the Web. A Web site can suddenly become extremely popular and receive far more requests in a relatively short time than it was originally configured to handle. In fact, a site may receive so many requests that it becomes "swamped," which typically renders it unusable. Besides making the one site inaccessible, heavy traffic destined to one location can congest the network near it, interfering with traffic at nearby sites.

As use of the Web has increased, so has the occurrence and impact of hot spots. Recent famous examples of hot spots on the Web include the JPL site after the Shoemaker-Levy 9 comet struck Jupiter, an IBM site during the Deep Blue-Kasparov chess tournament, and several political sites on the night of the election. In some of these cases, users were denied access to a site for hours or even days. Other examples include sites identified as "Web-site-of-the-day" and sites that provide new versions of popular software.

Our work was originally motivated by the problem of hot spots on the World Wide Web. We believe the tools we develop may be relevant to many client-server models, because centralized servers on the Internet such as Domain Name servers, Multicast servers, and Content Label servers are also susceptible to hot spots.

1.1 Past Work

Several approaches to overcoming the hot spots have been proposed. Most use some kind of replication strategy to store copies of hot pages throughout the Internet; this spreads the work of serving a hot page across several servers. In one approach, already in wide use, several clients share a *proxy cache*. All user requests are forwarded through the proxy, which tries to keep copies of frequently requested pages. It tries to satisfy requests with a cached copy; failing this, it forwards the request to the home server. The dilemma in this scheme is that there is more benefit if more users share the same cache, but then the cache itself is liable to get swamped.

Malpani et al. [6] work around this problem by making a group of caches function as one. A user's request for a page is directed to an arbitrary cache. If the page is stored there, it is returned to the user. Otherwise, the cache forwards the request to all other caches via a special protocol called "IP Multicast". If the page is cached nowhere, the request is forwarded to the home site of the page. The disadvantage of this technique is that as the number of participating caches grows, even with the use of multicast, the number of messages between caches can become unmanageable. A tool that we develop in this paper, *consistent hashing*, gives a way to implement such a distributed cache without requiring that the caches communicate all the time. We discuss this in Section 4.

Chankhunthod et al. [1] developed the Harvest Cache, a more scalable approach using a *tree* of caches. A user obtains a page by asking a nearby leaf cache. If neither this cache nor its siblings have the page, the request is forwarded to the cache's parent. If a page is stored by no cache in the tree, the request eventually reaches the root and is forwarded to the home site of the page. A cache

[*] This research was supported in part by DARPA contracts N00014-95-1-1246 and DABT63-95-C-0089, Army Contract DAAH04-95-1-0607, and NSF contract CCR-9624239

[1] Laboratory for Computer Science, MIT, Cambridge, MA 02139. email: {karger,e[redacted],danl,fil,mslevine,danl,rinap}@theory.lcs.mit.edu
A full version of this paper is available at: http://theory.lcs.mit.edu/~{karger,e[redacted],fil,mslevine,danl,rinap}
[2] Department of Mathematics, MIT, Cambridge, MA 02139

Distributed caching protocols for relieving hot spots on the World Wide Web

Out of the Tar Pit

Ben Moseley
ben@moseley.name

Peter Marks
public@indigomail.net

February 6, 2006

Abstract

Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional programming* and *Codd's relational model of data*.

1 Introduction

The "software crisis" was first identified in 1968 [NR69, p70] and in the intervening decades has deepened rather than abated. The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the handling of *state* and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are *code volume*, and explicit concern with the *flow of control* through the system.

The classical ways to approach the difficulty of state include object-oriented programming which tightly couples state together with related behaviour, and functional programming which — in its pure form — eschews state and side-effects all together. These approaches each suffer from various (and differing) problems when applied to traditional large-scale systems.

We argue that it is possible to take useful ideas from both and that — when combined with some ideas from the relational database world —

Functional programming to keep it simple

The Part-Time Parliament

Leslie Lamport

ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60]. | May 1998
ACM SIGOPS Hall of Fame Award in 2012

A fault-tolerant file system called Echo was built at SRC in the late 80s. The builders claimed that it would maintain consistency despite any number of non-Byzantine faults, and would make progress if any majority of the processors were working. As with most such systems, it was quite simple when nothing went wrong, but had a complicated algorithm for handling failures based on taking care of all the cases that the implementers could think of. I decided that what they were trying to do was impossible, and set out to prove it. Instead, I discovered the Paxos algorithm, described in this paper. At the heart of the algorithm is a three-phase consensus protocol. Dale Skeen seems to have been the first to have recognized the need for a three-phase protocol to avoid blocking in the presence of an arbitrary single failure. However, to my knowledge, Paxos contains the first three-phase commit algorithm that is a real algorithm, with a clearly stated correctness condition and a proof of correctness.

I thought, and still think, that Paxos is an important algorithm. Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island. Leo Guibas suggested the name Paxos for the island. I gave the Greek legislators the names of computer scientists working in the field, transliterated with Guibas's help into a bogus Greek dialect. (Peter Ladkin suggested the title.) Writing about a lost civilization allowed me to eliminate uninteresting details and indicate generalizations by saying that some details of the parliamentary protocol had been lost. To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist, replete with Stetson hat and hip flask.

My attempt at inserting some humor into the subject was a dismal failure. People who attended my lecture remembered Indiana Jones, but not the algorithm. People reading the paper apparently got so distracted by the Greek parable that they didn't understand the algorithm. Among the people I sent the paper to, and who claimed to have read it, were Nancy Lynch, Vassos Hadzilacos, and Phil Bernstein. A couple of months later I emailed them the following question:

Can you implement a distributed database that can tolerate the failure of any number of its processes (possibly all of them) without losing consistency, and that will resume normal behavior when more than half the processes are again working properly?

Paxos consensus algorithm for distributed systems

Raft

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support practical systems. As a result, both system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we set out to find a new consensus algorithm that could provide a better foundation for system building and education. Our approach was unusual in that our primary goal was *understandability*: could we define a consensus algorithm for practical systems and describe it in a way that is significantly easier to learn than Paxos? Furthermore, we wanted the algorithm to facilitate the development of intuitions that are essential for system builders. It was important not just for the algorithm to work, but for it to be obvious why it works.

The result of this work is a consensus algorithm called Raft. In designing Raft we applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations and is used by several companies; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

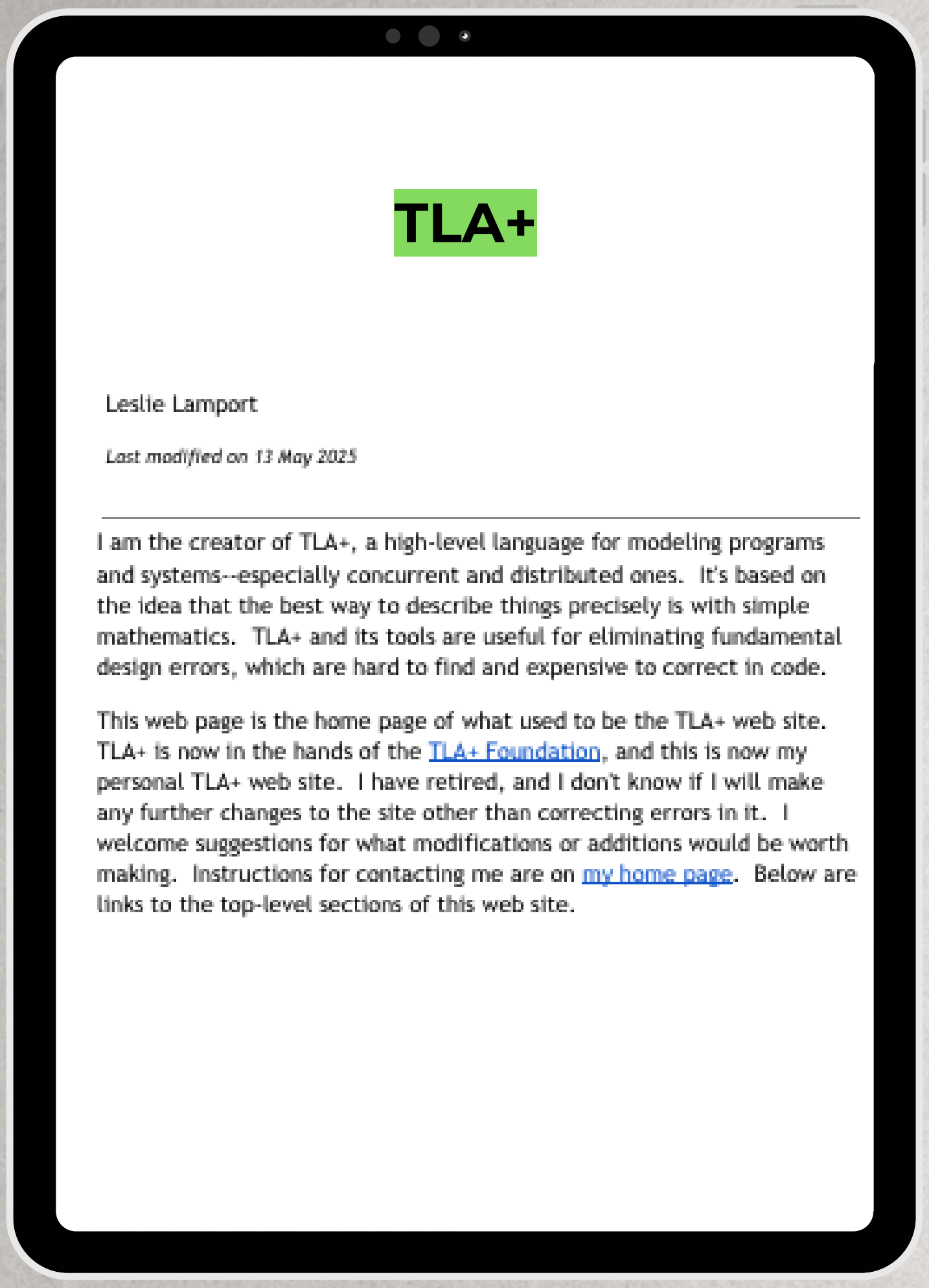
The remainder of the paper introduces the replicated state machine problem (Section 2), discusses the strengths and weaknesses of Paxos (Section 3), describes our general approach to understandability (Section 4), presents the Raft consensus algorithm (Sections 5–8), evaluates Raft (Section 9), and discusses related work (Section 10).

2 Replicated state machines

Consensus algorithms typically arise in the context of *replicated state machines* [37]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are

This tech report is an extended version of [32]; additional material is noted with a gray bar in the margin. Published May 20, 2014.

An understandable consensus algorithm



High-level language for modeling concurrent and distributed systems

Scaling Memcache at Facebook

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Abstract: Memcached is a well known, simple, in-memory caching solution. This paper describes how Facebook leverages memcached as a building block to construct and scale a distributed key-value store that supports the world's largest social network. Our system handles billions of requests per second and holds trillions of items to deliver a rich experience for over a billion users around the world.

1 Introduction

Popular and engaging social networking sites present significant infrastructure challenges. Hundreds of millions of people use these networks every day and impose computational, network, and I/O demands that traditional web architectures struggle to satisfy. A social network's infrastructure needs to (1) allow near real-time communication, (2) aggregate content on-the-fly from multiple sources, (3) be able to access and update very popular shared content, and (4) scale to process millions of user requests per second.

We describe how we improved the open source version of memcached [14] and used it as a building block to construct a distributed key-value store for the largest social network in the world. We discuss our journey scaling from a single cluster of servers to multiple geographically distributed clusters. To the best of our knowledge, this system is the largest memcached installation in the world, processing over a billion requests per second and storing trillions of items.

This paper is the latest in a series of works that have recognized the flexibility and utility of distributed key-value stores [1, 2, 5, 6, 12, 14, 34, 36]. This paper focuses on memcached—an open-source implementation of an in-memory hash table—as it provides low latency access to a shared storage pool at low cost. These qualities enable us to build data-intensive features that would otherwise be impractical. For example, a feature that issues hundreds of database queries per page request would likely never leave the prototype stage because it would be too slow and expensive. In our application, however, web pages routinely fetch thousands of key-value pairs from memcached servers.

One of our goals is to present the important themes that emerge at different scales of our deployment. While qualities like performance, efficiency, fault-tolerance, and consistency are important at all scales, our experience indicates that at specific sizes some qualities require more effort to achieve than others. For example, maintaining data consistency can be easier at small scales if replication is minimal compared to larger ones where replication is often necessary. Additionally, the importance of finding an optimal communication schedule increases as the number of servers increase and networking becomes the bottleneck.

This paper includes four main contributions: (1) We describe the evolution of Facebook's memcached-based architecture. (2) We identify enhancements to memcached that improve performance and increase memory efficiency. (3) We highlight mechanisms that improve our ability to operate our system at scale. (4) We characterize the production workloads imposed on our system.

2 Overview

The following properties greatly influence our design. First, users consume an order of magnitude more content than they create. This behavior results in a workload dominated by fetching data and suggests that caching can have significant advantages. Second, our read operations fetch data from a variety of sources such as MySQL databases, HDFS installations, and backend services. This heterogeneity requires a flexible caching strategy able to store data from disparate sources.

Memcached provides a simple set of operations (set, get, and delete) that makes it attractive as an elemental component in a large-scale distributed system. The open-source version we started with provides a single-machine in-memory hash table. In this paper, we discuss how we took this basic building block, made it more efficient, and used it to build a distributed key-value store that can process billions of requests per second. Hence-

Distributed cache architecture

Borg

Muhammad Tirmazi
Harvard University
tirmazi@g.harvard.edu

Adam Barker*
University of St Andrews
adb20@st-andrews.ac.uk

Nan Deng
Google
dengnan@google.com

Md E. Haque
Google
mehaque@google.com

Zhijing Gene Qin
Google
geneqin@google.com

Steven Hand
Google
sthand@google.com

Mor Harchol-Balter
CMU
harchol@cs.cmu.edu

John Wilkes
Google
johnwilkes@google.com

Zhijing Gene Qin
Google
geneqin@google.com

Steven Hand
Google
sthand@google.com

Mor Harchol-Balter
CMU
harchol@cs.cmu.edu

John Wilkes
Google
johnwilkes@google.com

Abstract

This paper analyzes a newly-published trace that covers 8 different Borg [35] clusters for the month of May 2019. The trace enables researchers to explore how scheduling works in large-scale production compute clusters. We highlight how Borg has evolved and perform a longitudinal comparison of the newly-published 2019 trace against the 2011 trace, which has been highly cited within the research community.

Our findings show that Borg features such as alloc sets are used for resource-heavy workloads; automatic vertical scaling is effective; job-dependencies account for much of the high failure rates reported by prior studies; the workload arrival rate has increased, as has the use of resource over-commitment; the workload mix has changed, jobs have migrated from the free tier into the best-effort batch tier; the workload exhibits an extremely heavy-tailed distribution where the top 1% of jobs consume over 99% of resources; and there is a great deal of variation between different clusters.

Keywords. Data centers, cloud computing.

ACM Reference Format:
Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3342195.3387517

1 Introduction

Data centers are huge capital investments that serve a wide range of applications including search engines, video processing, machine learning, and third-party cloud applications. Modern cluster management systems [22, 34, 35] have evolved to manage these data centers efficiently, and several companies have published job-scheduling traces of their cluster management systems so that external researchers can explore how they achieve this. Examples include Google [36], Microsoft [3] and Alibaba [33].

In 2011, Google published a 1-month trace from its Borg cluster management system to enable external researchers to explore how scheduling worked in a large-scale compute cluster. Several hundred researchers have taken advantage of that trace to study a wide range of phenomena. An open question is how workloads for such systems evolve over time, and how the evolution of the cluster managers has affected scheduling decisions. To support research on these topics, Google has published a new "2019" trace [37] that includes detailed Borg job scheduling information from 8 different Google compute clusters (Borg cells) for the entire month of May 2019. The new trace contents are an extension of the data provided in the 2011 trace [28, 36]. This paper reports on the results of a first longitudinal comparison between the 2011 and 2019 traces, repeating several analyses from the first paper to analyze the 2011 trace [27], and adding some new ones. We also discuss new features in Borg that were not visible in the 2011 trace. Our key observations can be summarized as follows:

1. **The 2019 trace has several new features (§3 and §5)**, including information about allocations [35], parent-child dependencies (which affects how task exits are to

Date	*May 2011*	*May 2019*
Duration	30 days	31 days
Cells (clusters)	1	**8**
Machines	12.6k	**96.4k**
Machines per cell	12.6k	12.0k
Hardware platforms	3	7
Machine shapes	10	**21**
Priority values	0–11	**0–450**
Alloc sets	–	**Y**
Job dependencies	–	**Y**
Batch queueing	–	**Y**
Vertical scaling	–	**Y**
Format	csv files	BigQuery tables

Table 1. Comparison between 2011 and 2019 traces.

*Work done whilst on sabbatical leave as a Visiting Researcher at Google.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04.
https://doi.org/10.1145/3342195.3387517

Google's cluster management system

The Next 700 Programming Languages

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a sequential (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncrasies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter rather than the former. The conclusion follows that many language characteristics are irrelevant to the alleged problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after more syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation, and if the dialects are well disciplined, they might with luck be characterized as differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have folded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(y+a)$	$f(3+2c)$
where $x = b + 2c$	**where** $f(x) = x(y+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication, these uses of '**where**' require no explanation. Nor do the following:

$f(b+2c) + f(2b-c)$
where $f(x) = x(y+a)$

$f(b+2c) + f(2b-c)$
where $f(x) = x(y+a)$
and $b = u/(v+1)$
and $c = v/(v+1)$

$g(f$ **where** $f(x) = ax^2 + bx + c,$
$u/(v+1),$
$v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

[1] There is no more use or mention of λ in this paper—cognoscenti will nevertheless sense an undercurrent. A not inappropriate title would have been "Church without Lambda."

A universal framework for language design and implementation

Dynamo

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Amazon's Highly Available Key-value Store

Learn from smart engineers
by reading white papers on years
of their work condensed into a few pages.

What else should make this list?

If you enjoyed this...

You'll love my Playbook.

Click the link in my bio
to join my newsletter, and
download my Playbook for free.

newsletter.systemdesign.one