

# Cloud Computing and Big Data

Consensus



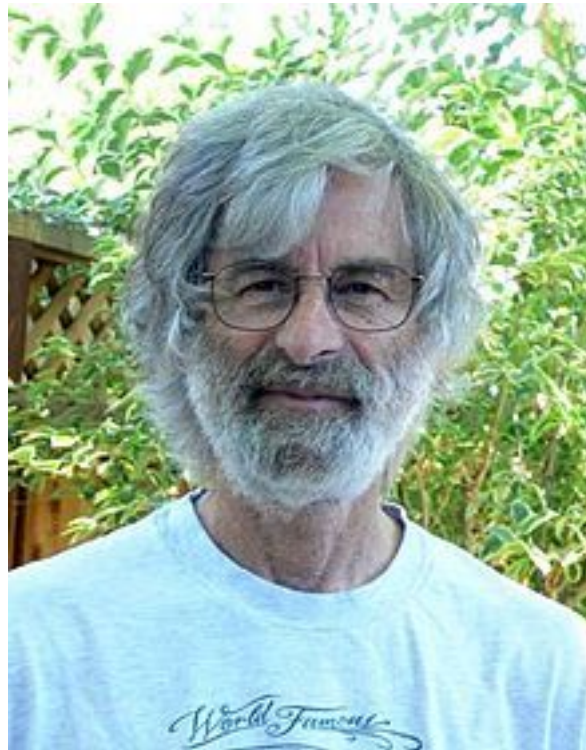
© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License  
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

# Fundamental problems in Distributed Computing

- Efficient distribution of work
  - combating *serialization*
- Consensus
  - combating *failure*



# Leslie Lamport



© Paul Fremantle 2015. This work is licensed under a Creative Commons  
Attribution-NonCommercial-ShareAlike 4.0 International License  
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

# Consensus

- In almost any cluster of machines
  - Spark, Kafka, Cassandra, etc
- We need:
  - A consistent configuration
  - (often) A controller
- The controller may fail
  - Therefore we need to be able to detect that and elect a new leader/controller
- “Crash Fault Tolerance”





© Paul Fremantle 2015. This work is licensed under a Creative Commons  
Attribution-NonCommercial-ShareAlike 4.0 International License  
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

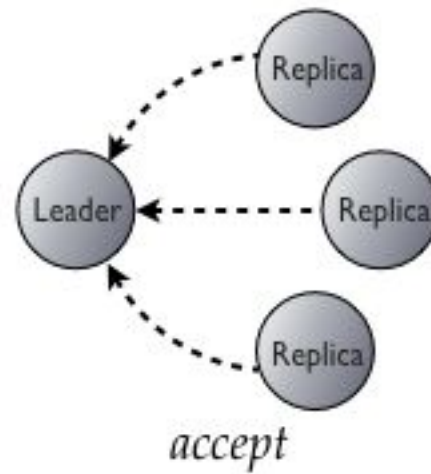
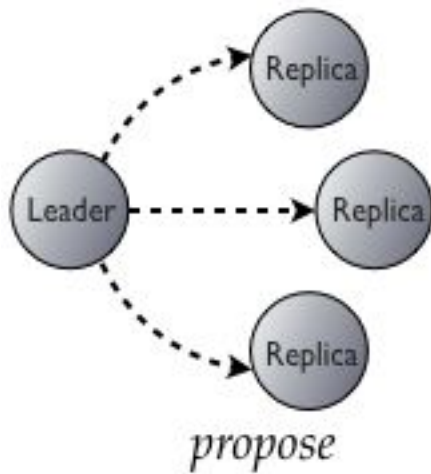
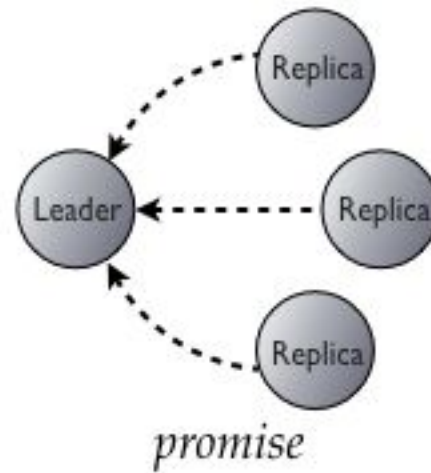
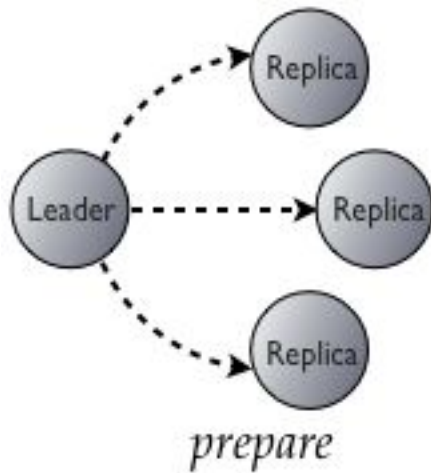


# The Part-Time Parliament

Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.





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?

None of them noticed any connection between this question and the Paxos algorithm.

I submitted the paper to *TOCS* in 1990. All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed. I was quite annoyed at how humorless everyone working in the field seemed to be, so I did nothing with the paper. A number of years later, a couple of people at SRC needed algorithms for distributed systems they were building, and Paxos provided just what they needed. I gave them the paper to read and they had no problem with it. Here is Chandu Thekkath's account of the history of Paxos at SRC.





# Paxos in production

- Cassandra uses Paxos for leadership election
- Google Chubby (distributed lock service) and hence BigTable
- Google Spanner and Megastore
- OpenReplica
- IBM SAN Volume Controller
- Microsoft Autopilot cluster management
- WANdisco have implemented Paxos within DConE
- XtremFS uses a Paxos-based lease negotiation algorithm
- Heroku uses Doozerd consistent distributed data store.
- Ceph uses Paxos
- The Clustrix distributed SQL database uses Paxos for distributed transaction resolution.
- Neo4j HA graph database Paxos, replacing ZooKeeper



# 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.

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 timeouts to



# Raft consists of two main parts

- Leadership elections
- Creating a consistent log once a leader is in place



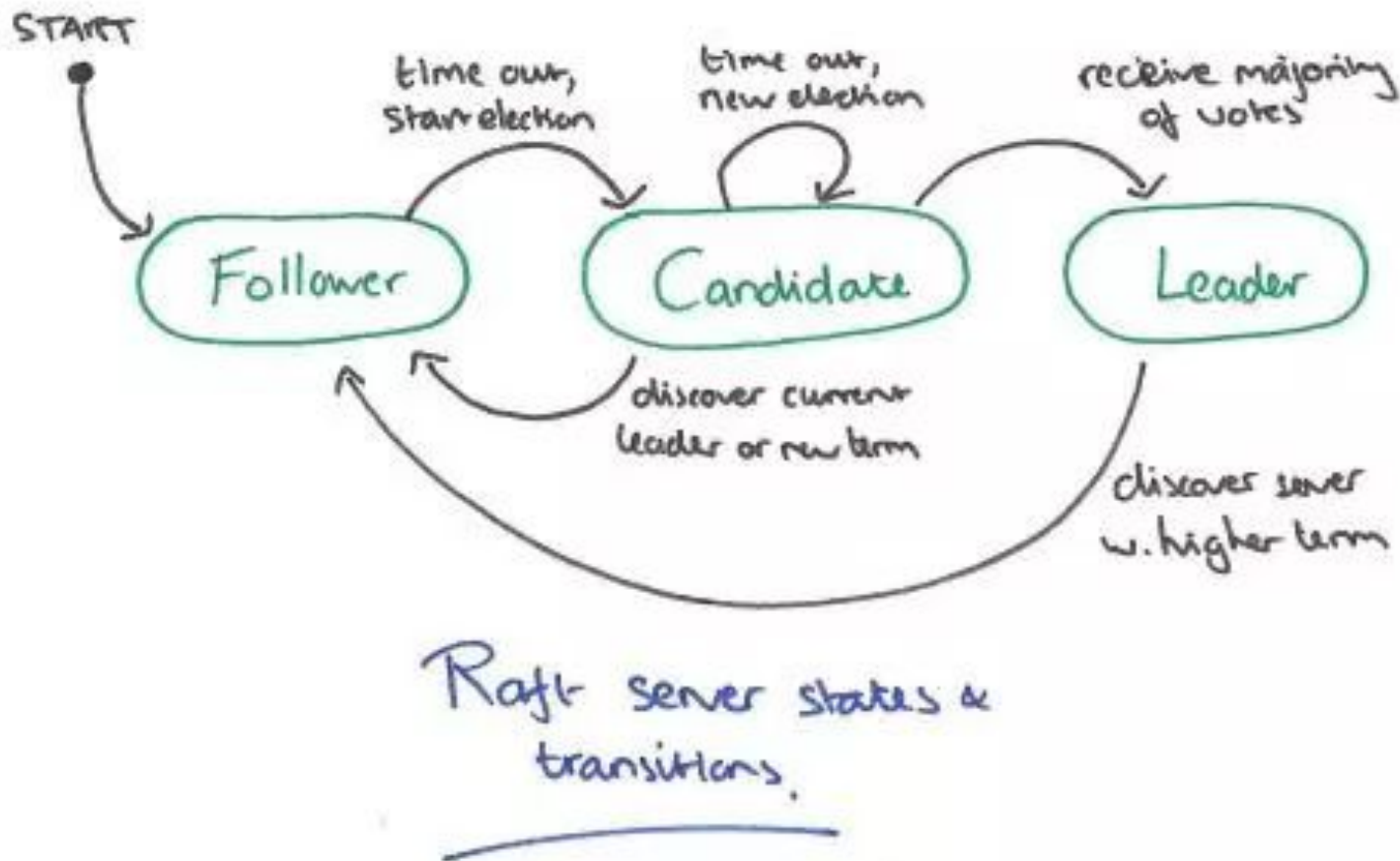
# Raft Key concepts

- Leader
  - The leader ingests new log entries, distributes to peers and decides when they are committed.
- Quorum
  - $(n \div 2) + 1$  members, where  $n$  is the size of the peer set
- Peer Set
  - The set of nodes participating
- Committed Entry
  - An entry is committed if it is stored persistently on a quorum of nodes. Then it can be *applied*.
- Log
  - The log is consistent iff all participants agree on the entries and the order
- Finite State Machine
  - A Finite State Machine is well defined set of states and transitions. Ensures that each peer applying the same logs reaches the same conclusions





# Raft Leadership Election



<https://blog.acolyer.org/2015/03/12/in-search-of-an-understandable-consensus-algorithm/>

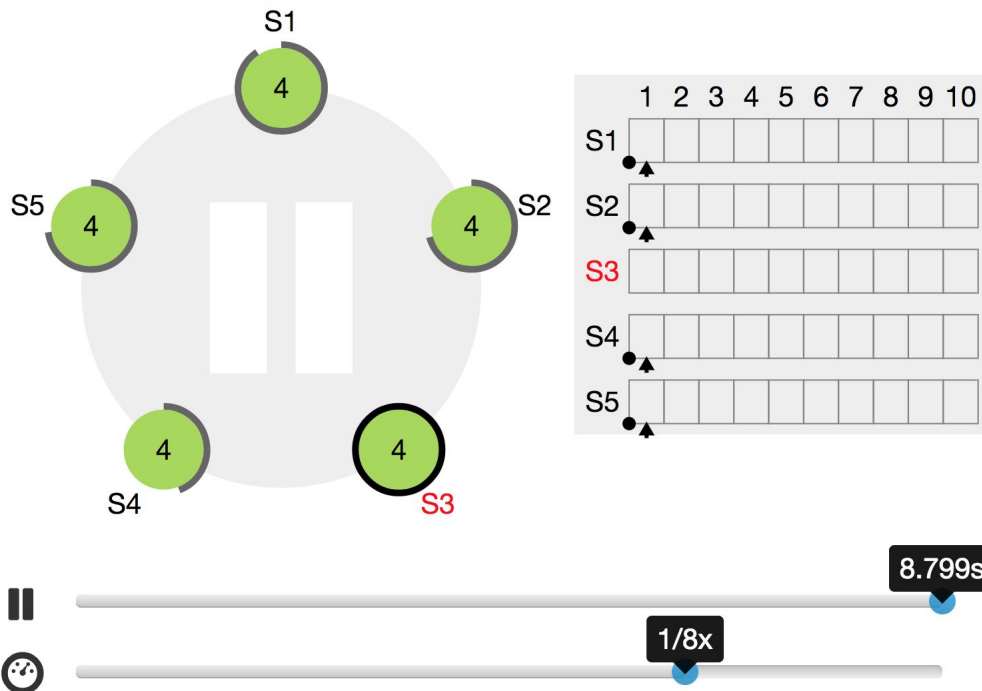
# Once a leader is in place

- Client requests the leader writes a log entry
- The leader writes it to durable storage
- Replicates to a quorum of followers
- Once the log entry is committed, it can be applied
  - Execute the FSM with the new data



# Nice visualisations of Raft

- <http://thesecretlivesofdata.com/raft/>
- <https://raft.github.io/>



# Implementations / Users of Raft

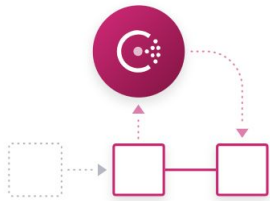
- etcd
- Consul
- TiKV
- OpenDaylight Controller
- CopyCat from Atomix
- JGroups





# Consul

## Use Cases



### Service Discovery

for connectivity

Service Registry enables services to register and discover each other.

[Learn more](#)

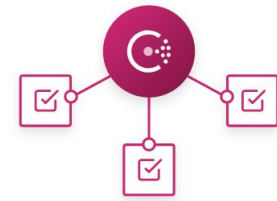


### Service Segmentation

for security

Secure service-to-service communication with automatic TLS encryption and identity-based authorization.

[Learn more](#)



### Service Configuration

for runtime configuration

Feature rich Key/Value store to easily configure services.

[Learn more](#)

TiKV (The pronunciation is: /'taɪkɜːvi/ tai-K-V, etymology: titanium) is a distributed Key-Value database powered by Rust and Raft. TiKV is based on the design of Google Spanner and HBase, but is much simpler without dependency on any distributed file system.

With the implementation of the Raft consensus algorithm in Rust and consensus state stored in RocksDB, TiKV guarantees data consistency. [Placement Driver \(PD\)](#), which is introduced to implement sharding, enables automatic data migration. The transaction model is similar to Google's Percolator with some performance improvements. TiKV also provides snapshot isolation (SI), snapshot isolation with lock (SQL: `SELECT ... FOR UPDATE`), and externally consistent reads and writes in distributed transactions.

TiKV has the following key features:

- **Geo-Replication**

TiKV uses [Raft](#) and the Placement Driver to support Geo-Replication.

- **Horizontal scalability**

With PD and carefully designed Raft groups, TiKV excels in horizontal scalability and can easily scale to 100+ TBs of data.

- **Consistent distributed transactions**

Similar to Google's Spanner, TiKV supports externally-consistent distributed transactions.

- **Coprocessor support**

Similar to Hbase, TiKV implements a coprocessor framework to support distributed computing.



# Zab: High-performance broadcast for primary-backup systems

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini

Yahoo! Research

{fpj,breed,serafini}@yahoo-inc.com

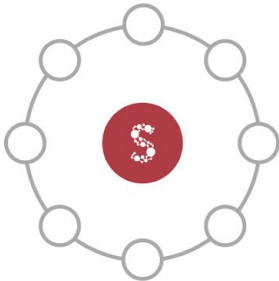
**Abstract**—Zab is a crash-recovery atomic broadcast algorithm we designed for the ZooKeeper coordination service. ZooKeeper implements a primary-backup scheme in which a primary process executes clients operations and uses Zab to propagate the corresponding incremental state changes to backup processes<sup>1</sup>. Due the dependence of an incremental state change on the sequence of changes previously generated, Zab must guarantee that if it delivers a given state change, then all other changes it depends upon must be delivered first. Since primaries may crash, Zab must satisfy this requirement despite crashes of primaries.

Applications using ZooKeeper demand high-performance from the service, and consequently, one important goal is the ability of having multiple outstanding client operations at a time. Zab enables multiple outstanding state changes by guaranteeing that at most one primary is able to broadcast state changes and have them incorporated into the state, and by using a synchronization phase while establishing a new primary. Before this synchronization phase completes, a new primary does not broadcast new state changes. Finally, Zab uses an identification scheme for state changes that enables a process to easily identify missing changes. This feature is key for efficient recovery.

scheme [5], [6], [7] to maintain the state of replica processes consistent. With ZooKeeper, a primary process receives all incoming client requests, executes them, and propagates the resulting non-commutative, incremental state changes in the form of *transactions* to the backup replicas using *Zab*, the ZooKeeper atomic broadcast protocol. Upon primary crashes, processes execute a recovery protocol both to agree upon a common consistent state before resuming regular operation and to establish a new primary to broadcast state changes. To exercise the primary role, a process must have the support of a quorum of processes. As processes can crash and recover, there can be over time multiple primaries and in fact the same process may exercise the primary role multiple times. To distinguish the different primaries over time, we associate an instance value with each established primary. A given instance value maps to at most one process. Note that our notion of instance shares some of the properties of views of group communication [8] but it presents some key differences. With



# Serf.io

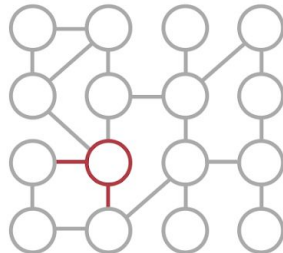


## Gossip-based Membership

Serf relies on an efficient and lightweight gossip protocol to communicate with nodes. The Serf agents periodically exchange messages with each other in much the same way that a zombie apocalypse would occur: it starts with one zombie but soon infects everyone. In practice, the gossip is **very fast and extremely efficient**.

## Failure Detection

Serf is able to quickly detect failed members and notify the rest of the cluster. This failure detection is built into the heart of the gossip protocol used by Serf. Like humans in a zombie apocalypse, everybody checks their peers for infection and quickly alerts the other living humans. Serf relies on a random probing technique which is proven to efficiently scale to clusters of any size.





# SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol

Abhinandan Das, Indranil Gupta, Ashish Motivala\*  
Dept. of Computer Science, Cornell University  
Ithaca NY 14853 USA  
{`asdas, gupta, ashish`}@`cs.cornell.edu`

## Abstract

*Several distributed peer-to-peer applications require weakly-consistent knowledge of process group membership information at all participating processes. SWIM is a generic software module that offers this service for large-scale process groups. The SWIM effort is motivated by the unscalability of traditional heart-beating protocols, which either impose network loads that grow quadratically with group size, or compromise response times or false positive frequency w.r.t. detecting process crashes. This paper reports on the design, implementation and performance of the SWIM sub-system on a large cluster of commodity PCs.*

*Unlike traditional heartbeating protocols, SWIM separates the failure detection and membership update dissemination functionalities of the membership protocol. Processes are monitored through an efficient peer-to-peer periodic randomized probing protocol. Both the expected time to first detection of each process failure, and the expected message load per member, do not vary with group size.*

## 1. Introduction

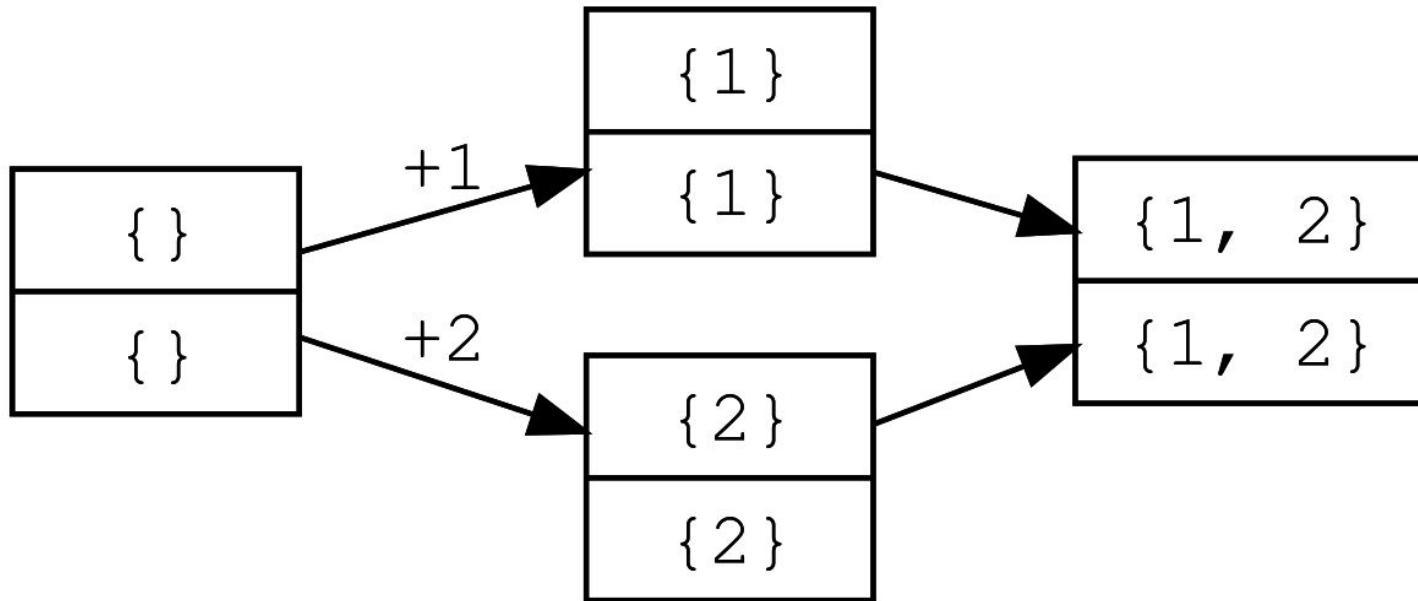
*As you swim lazily through the milieu,  
The secrets of the world will infect you.*

Several large-scale peer-to-peer distributed process groups running over the Internet rely on a distributed membership maintenance sub-system. Examples of existing middleware systems that utilize a membership protocol include reliable multicast [3, 11], and epidemic-style information dissemination [4, 8, 13]. These protocols in turn find use in applications such as distributed databases that need to reconcile recent disconnected updates [14], publish-subscribe systems, and large-scale peer-to-peer systems[15]. The performance of other emerging applications such as large-scale cooperative gaming, and other collaborative distributed applications, depends critically on the reliability and scalability of the membership maintenance protocol used within.

Briefly, a membership protocol provides each process (“member”) of the group with a locally-maintained list of other non-faulty processes in the group. The protocol en-

# Conflict-free Replicated Data Types

- Also known as {Commutative, Convergent, Confluent}



weaveworks/mesh: A tool for b x Viewstamped replication: A ne x Zab: High-performance broad x Viewstamped Replication Revi x Paul

← → ↻ 🏠 GitHub, Inc. [US] | https://github.com/weaveworks/mesh ☆

Apps 📧 Inbox (23,972) - pa... 📅 WSO2, Inc. - Calen... 🌐 Inbox (272,501) - p... 📧 Inbox (21,796) - pa... 📧 "A: To me" (587) - ... 📅 Fremantle Family -... 📄 Google Docs » 📁 Other Bookmarks

# mesh

godoc reference **PASSED**

---

Mesh is a tool for building distributed applications.

Mesh implements a [gossip protocol](#) that provide membership, unicast, and broadcast functionality with [eventually-consistent semantics](#). In CAP terms, it is AP: highly-available and partition-tolerant.

Mesh works in a wide variety of network setups, including thru NAT and firewalls, and across clouds and datacenters. It works in situations where there is only partial connectivity, i.e. data is transparently routed across multiple hops when there is no direct connection between peers. It copes with partitions and partial network failure. It can be easily bootstrapped, typically only requiring knowledge of a single existing peer in the mesh to join. It has built-in shared-secret authentication and encryption. It scales to on the order of 100 peers, and has no dependencies.

## Using

---

Mesh is currently distributed as a Go package. See [the API documentation](#).

We plan to offer Mesh as a standalone service + an easy-to-use API. We will support multiple deployment scenarios, including as a standalone binary, as a container, as an ambassador or [sidecar](#) component to an existing container, and as an infrastructure service in popular platforms.

## Developing

---

Mesh builds with the standard Go tooling. You will need to put the repository in Go's expected directory structure; i.e.,

```
$GOPATH/src/github.com/weaveworks/mesh .
```

## Building

If necessary, you may fetch the latest version of all of the dependencies into your GOPATH via





# The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE

SRI International

---

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

---

## 1. INTRODUCTION

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem. We devote the major part of the paper to a discussion of this abstract problem and conclude by indicating how our solutions can be used in implementing a reliable computer system.



# Byzantine failures

- A Byzantine fault is any fault presenting different symptoms to different observers.
- Imagine a set of generals trying to formulate a plan
  - Some of the generals are traitors
  - They may lie to other generals
  - They may lie selectively



## Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov  
*Laboratory for Computer Science,  
Massachusetts Institute of Technology,  
545 Technology Square, Cambridge, MA 02139*  
`{castro,liskov}@lcs.mit.edu`

### Abstract

This paper describes a new replication algorithm that is able to tolerate Byzantine faults. We believe that Byzantine-fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior. Whereas previous algorithms assumed a synchronous system or were too slow to be used in practice, the algorithm described in this paper is practical: it works in asynchronous environments like the Internet and incorporates several important optimizations that improve the response time of previous algorithms by more than an order of magnitude. We implemented a Byzantine-fault-tolerant NFS service using our algorithm and measured its performance. The results show that our service is only 3% slower than a standard unreplicated NFS.

and replication techniques that tolerate Byzantine faults (starting with [19]). However, most earlier work (e.g., [3, 24, 10]) either concerns techniques designed to demonstrate theoretical feasibility that are too inefficient to be used in practice, or assumes synchrony, i.e., relies on known bounds on message delays and process speeds. The systems closest to ours, Rampart [30] and SecureRing [16], were designed to be practical, but they rely on the synchrony assumption for correctness, which is dangerous in the presence of malicious attacks. An attacker may compromise the safety of a service by delaying non-faulty nodes or the communication between them until they are tagged as faulty and excluded from the replica group. Such a denial-of-service attack is generally easier than gaining control over a non-faulty node.

# Summary

- Distributed consensus algorithms existed since the 1990s
- More recently gaining traction
- A key part of big data systems
  - Leadership election
  - Consistent configuration

