

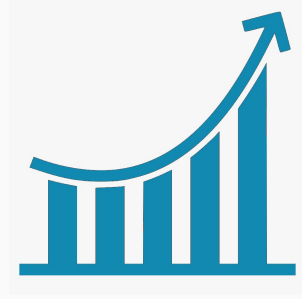
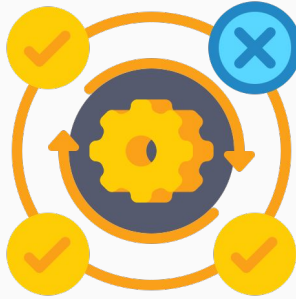
# Build Your Own Distributed System Using Go

GopherCon 2023  
Philip O'Toole  
[www.philipotoole.com](http://www.philipotoole.com)



Hello everyone and welcome to Build Your Own Distributed System using Go. My name is Philip and it's great to be here to speak with you all today.

# Why Distributed Systems matter



Philip O'Toole  
<https://www.philipotoole.com>

Before we dive into who I am and what we're doing here, let's start with a question:  
**Why should you care about distributed systems today?**

Distributed systems are the backbone of modern computing -- especially web-scale computer. From the databases that power our favorite applications to the real-time analytics tools many companies rely on—distributed systems make it possible.

In a world that demands **high availability, fault tolerance, and scalability** of its computer systems, us programmers, developers and computer scientists long ago decided Distributed Systems were the answer.

Some of the software and techniques I will describe today is at the center of some of the most popular software today, including Kubernetes and Hashicorp Consul. And even if -- as many of you will -- decide to run someone else's distributed system, understanding how they are built will make you a better programmer, designer, and operator of those very same systems.

# Key takeaways

- Understand why Distributed Systems present unique challenges – and how the Raft consensus protocol addresses them
- Learn to integrate Raft with different data storage systems through practical examples
- Look at some practicalities of developing, testing, and managing your Distributed System

Philip O'Toole  
<https://www.philipotoole.com>

**First of all** there will be a slide at the end of my talk with a link to this deck – that's the slide to take a picture of. You'll be able to download the entire deck, and my speaker notes, so don't worry about taking notes (unless you really want to). I've also got a fair amount to get through today, and will try to make time for any questions at the end. But I'll also stay around after the presentation, and answer any questions folks may have.

**Now, let's look at what I hope you will take away from today's talk.** We will:

My intent today is to impart practical knowledge, drawn from hands-on experience, to assist you in constructing your own distributed systems.

## About me



**rqlite.io**

*The lightweight, distributed relational  
database built on Raft and SQLite*

Philip O'Toole  
<https://www.philipotoole.com>

**I am Philip, a software engineer with a focus on Go and distributed databases.**

I am the creator rqlite, a lightweight distributed relational database built on Raft and SQLite.

I also spent two years as part of the core team at InfluxDB, a distributed time-series database that also leverages Raft. Currently, I work at Google, managing development teams working on large-scale logging systems for Google Cloud Platform. **And I am here today in a personal capacity, not as representative of Google, nor any other company.**

# Why Distributed Systems are hard



Why are Distributed Systems difficult?

There really is only one hard problem in distributed systems.

And to gain an understanding of what we're going to build today, it's important to gain an intuitive understanding of what problem we actually have to solve.

# Why Distributed Systems are hard



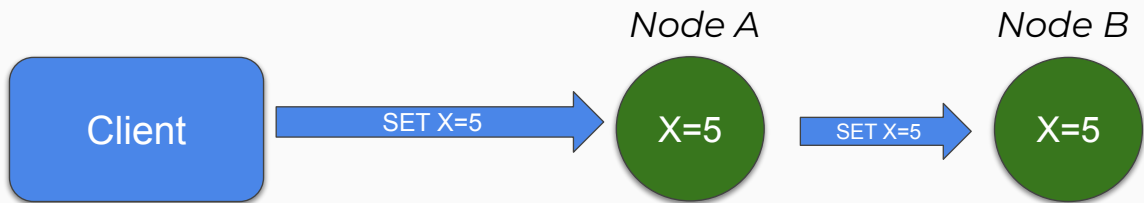
Philip O'Toole  
<https://www.philipotoole.com>

Let's say you want to set a piece data to a particular value. For example it could be a row in a database table. It could be the password for a user. It could -- every simpler -- be a value for key in a key-value store.

That's easy, we all know how to do it. Make the call to your system -- in the example above, let's set X to 5.

But what if the data you're setting is really valuable? What if it's mission critical, and you need multiple copies of that data, in separate places? What if you need that data to be always available for retrieval by others, even in the face of failure? Then we need to make copies -- we need to **replicate** the data.

# Why Distributed Systems are hard

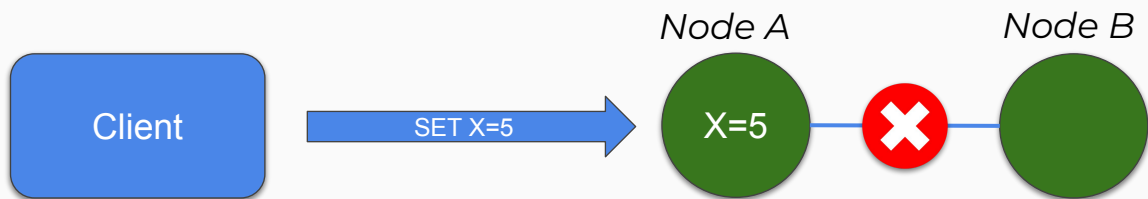


Philip O'Toole  
<https://www.philipotoole.com>

So let's say we build a system that makes that copy in the background. Great! This solves the problem -- or at least, appears to.

But what happens if Node B doesn't respond? Should Node A respond OK to the client? Should it respond with an error? What if Node B got the update, but the connection between the nodes failed before Node B could acknowledge the copy? See? It starts to get complicated.

# Why Distributed Systems are hard



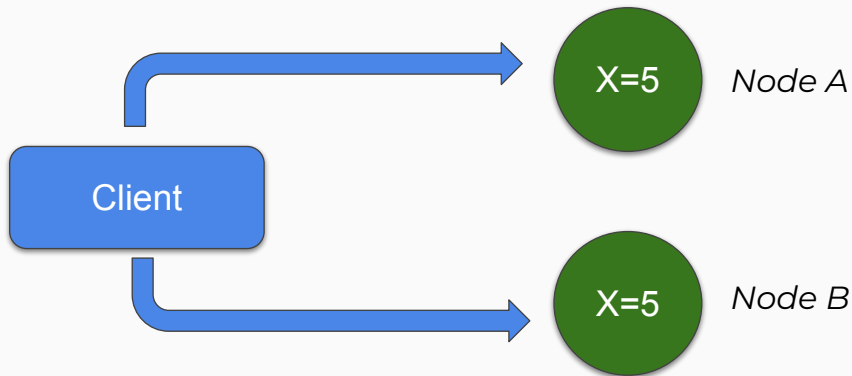
This failure is known as a ***partition***.

Philip O'Toole  
<https://www.philipotoole.com>

In other words, what should happen if this system suffers from a partition?



# Why Distributed Systems are hard



Client replication: multi-node system - each node must support changing state

Philip O'Toole  
<https://www.philipotoole.com>

OK, so say you decide to try a different approach.

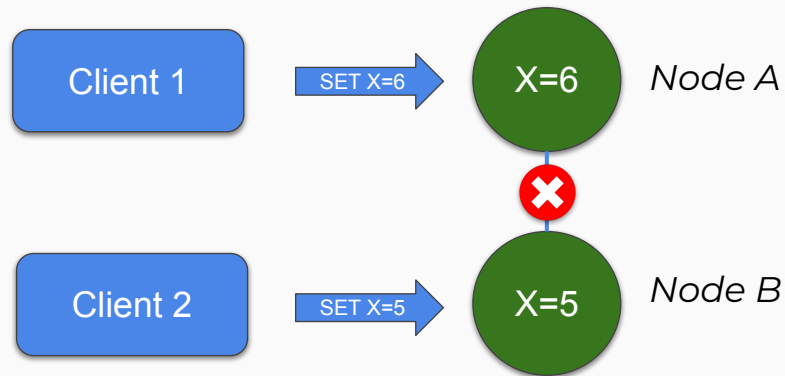
Another idea is to push the responsibility to the client.

So now the client is responsible for replicating the data.

This means more work for the client, but does give the client maximum flexibility -- and information about what has happened at each stage.

But there is a subtlety introduced here -- unlike the first system each node in a cluster must be able to receive updates from the client.

# Why Distributed Systems are hard



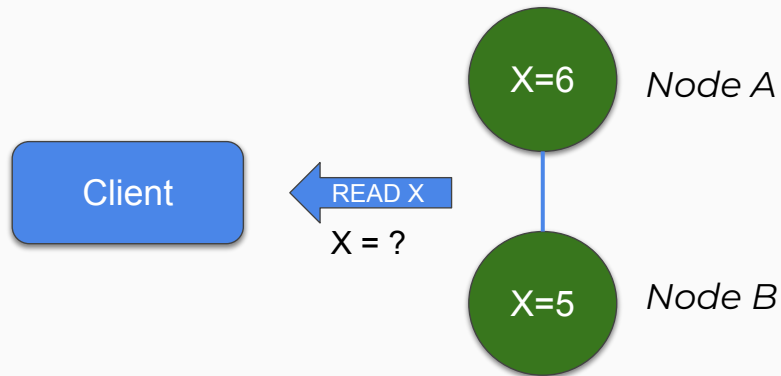
Client replication: multi-node system - each node must support changing state

Philip O'Toole  
<https://www.philipotoole.com>

And if each node can accept updates, then you must solve the problem that can occur if conflicting updates are received by different nodes! And if there is a partition, what the system will do is even harder to control.

Because what happens after this, if there is a read?

# Why Distributed Systems are hard



What value should be returned for X?

Philip O'Toole  
<https://www.philipotoole.com>

And when the cluster is repaired, what is the resolution mechanism?

Now **there are solutions to these issues**, but my point is to **show there is a real problem here**, and solving this **apparent simple request** -- replicate my data reliably, consistently, and quickly -- is actually harder to solve than you might realise.

# Why Distributed Systems are hard

This problem is known as  
*Distributed Consensus.*

Philip O'Toole  
<https://www.philipotoole.com>

This problem is known as *Distributed Consensus*. Decades of computer science research have gone into solving it.

And we have solved it.

# Introducing Raft

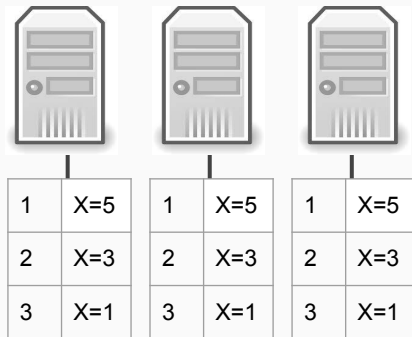
*Raft* is a protocol for solving the  
*Distributed Consensus* problem

Philip O'Toole  
<https://www.philipotoole.com>

What is Raft?

Raft appeared in 2014 and was created by [Diego Ongaro](#) and [John Ousterhout](#) at Stanford. One of its main goals was to create a protocol that was relatively easy to understand.

# Core Raft concepts



An agreed upon sequence of events, on every node - the *Raft log*.



Raft defines *Snapshotting* -- how we ensure the Raft Log doesn't grow without bound

Philip O'Toole  
<https://www.philipotoole.com>

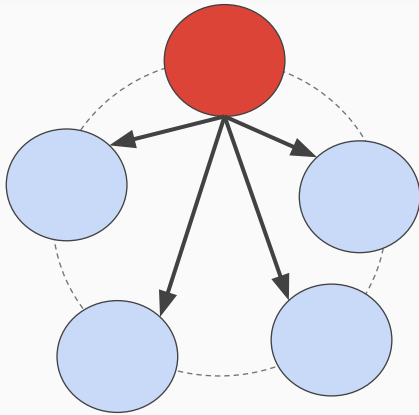
By definition, every node in a Raft cluster agrees on the events, and sequence of those events, in what is called Raft Log.

In fact, the Raft consensus protocol is **specifically** about ensuring that the log is the same on every node. All other functionality, flows from this fact. Raft has nothing to say about what is done with that log. What we will actually do is apply those events -- in sequence -- to our storage system.

Raft also defines a process known as Snapshotting? What is Snapshotting? Well it will help if you first understand what problem Snapshotting is designed to solve.

If you think about it, if the Raft Log contains every single event since the beginning of time (so to speak) then what is to prevent the Raft log from growing without bound? That is where Snapshotting comes in. Every so often the Raft system will ask your code to provide a snapshot -- and encapsulation of the entire state of your storage system -- and then it will remove from the log all entries which are represented in that state machine. That is how we avoid having the log grow without bound.

# Core Raft concepts



*Leader nodes - and process for selecting that *Leader*.*



*Heartbeating between nodes to detect failure*

Philip O'Toole  
<https://www.philipotoole.com>

Raft also defines the Leader election protocol, which ensures there is always one, and only one, Leader. It is the Leader which is responsible for coordinating changes to the log on each node. Nodes which are not the Leader are called **Followers**.

Raft also defines how to detect if a node fails, via a mechanism called "Heartbeating".

# What Raft solves – and how

| Concurrent updates  | Consistency   | Fault Tolerance  |
|---|---|--|
| <ul style="list-style-type: none"><li>• <b>Only</b> the Leader can make changes</li></ul> | <ul style="list-style-type: none"><li>• Raft <b>log</b> is key</li><li>• No “eventually consistent” results</li></ul> | <ul style="list-style-type: none"><li>• Only a <b>quorum</b> need agree</li><li>• Failure-detection via <b>Heartbeating</b></li><li>• <i>Leader Election</i></li></ul> |

Philip O'Toole  
<https://www.philipotoole.com>

Let's take a closer look at how each of these characteristics address the hard problem of Distributed Systems

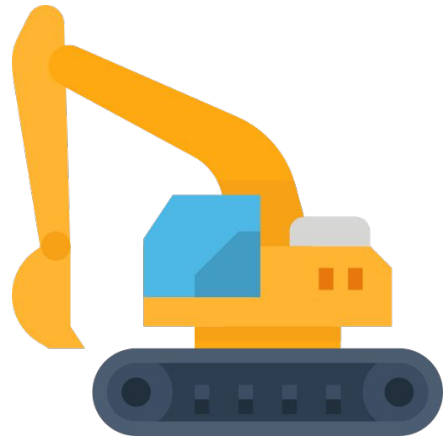
Concurrent updates  
Consistency  
Fault tolerance

In summary Raft provides a solution for the distributed consensus problem, abstracting away the details, allowing programmers like us to choose the state to be agreed on by those nodes.

I'm not going into the details on how Raft does all this, but it's conceptually rather simple and I have links to resources at the end of my talk. What I want to focus on is how you can take this primitive and apply it to your storage system.



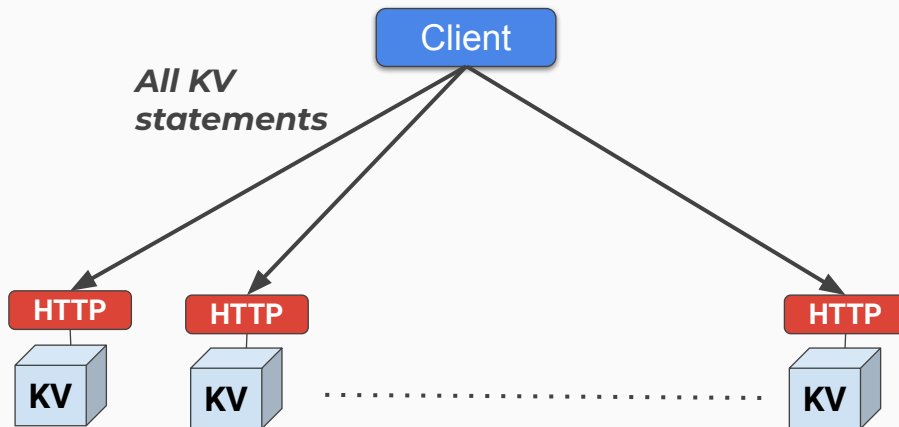
# Let's build



Now that we've established why you need Raft, let's look at the practicalities of building a system that uses it.

Let's walk through the conversion of a simple key-value to a distributed version. Many talks showing how to use Raft take a bottom-up approach, showing Raft code from the start, and explaining how it works. I'm going to take a different approach – instead I want you to develop an intuitive sense for what is going on with the system, where Raft **as a building block** fits in, and show you the key code changes you make to change a simple key-value store into a distributed version.

# Naive Distributed Key-Value store



Philip O'Toole  
<https://www.philipotoole.com>

Let's look at the system diagram for a simple key-value store, one where the client attempts to do its own replication. Of course, this system suffers from all the problems we outlined earlier. For example it's not clear what the client should do if some KV stores don't respond, or how the system should operate if multiple clients are writing to the KV stores simultaneously.

Let's next take a look at very simple Go code, which implements one of these KV store nodes.

# Naive Distributed Key-Value store

```
curl -XPOST localhost:8080/key -d '{"user1": "batman"}'
```

```
// KVService is a KV store which responds to HTTP requests
type KVService {
    kv map[string]string
}

func (s *KVService) SetKey(w http.ResponseWriter, r *http.Request) {
    buf := io.ReadAll(r.Body) // Read the request body

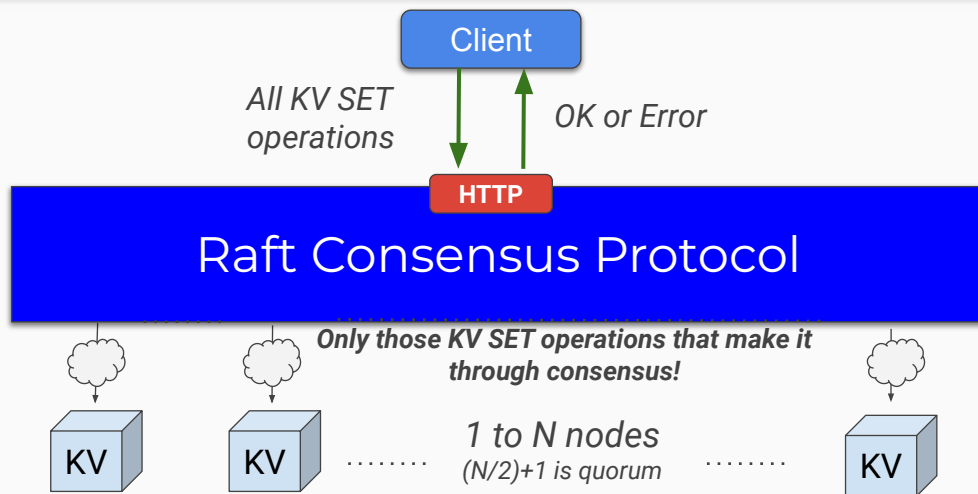
    m := map[string]string{} // Actually set values in KV store
    json.Unmarshal(buf, &m)
    for k, v := range m {
        s.kv[k] = v
    }
}
```



Philip O'Toole  
<https://www.philipotoole.com>

Now let's show how adding Raft to our naive distributed KV store changes the system design.

# Raft-based Key-Value store - conceptual view



Philip O'Toole  
<https://www.philipotoole.com>

Conceptually, this is what we're going to do. Now it's important to understand that I don't consider this a system architecture diagram. Instead this is a conceptual diagram, showing what is actually happening in the distributed KV store we're about to build. It is important to note that this system is full replicated. **A full copy of the KV data lives on every node. This is a distributed system -- it is replicating for high availability and fault tolerance, but not for write performance.**

Clients now write to a Raft "black box". This black box accepts operations – mutations if you will – and takes care of pushing them out to the cluster. It tracks which nodes have received the changes, if quorum has been met, and anything else that needs to happen to ensure the change is safely replicated. Only when all this has happened does it respond back to the client. From the client's point-of-view, nothing has changed – except it's much simpler to work with the KV store!

So how do we go about actually adding this Raft building block to our KV store? Let's start by introducing one of the best open-source Raft Go implementations out there – Hashicorp's Raft library.

# Choosing a Raft Consensus Library

## HashiCorp Raft

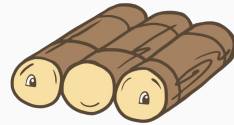
Mature implementation in Go

Powers Consul and Nomad

Good documentation

Customizable

MIT Licensed\*



Philip O'Toole  
<https://www.philipotoole.com>

- Available on GitHub at <https://github.com/hashicorp/raft>
- \*Version 1.5 is MIT licensed, though future versions are subject to change. It's important to note this because Hashicorp recently changed the licensing for their \*products\*, but not for libraries such as the Raft library.

# Building a Raft-based Key-Value store




```
// KVService is a distributed Key-Value store which uses Raft
type KVService {
    kv map[string]string // Each Service still has a KV store
    consensus *raft.Raft // Each Service now has a Raft module too!
}

// SetKey sets the value of key via Raft Distributed Consensus
func (s *KVService) SetKey(w http.ResponseWriter, r *http.Request) {
    buf := io.ReadAll(r.Body) // Read the request body
    m := map[string]string{}
    json.Unmarshal(buf, &m)
    for k, v := range m {
        s.kv[k] = v
    }
    future := s.consensus.Apply(buf, 5 * time.Second) // Call Raft!
    if future.Error() != nil {
        w.WriteHeader(http.StatusServiceUnavailable)
    }
}
```

Philip O'Toole

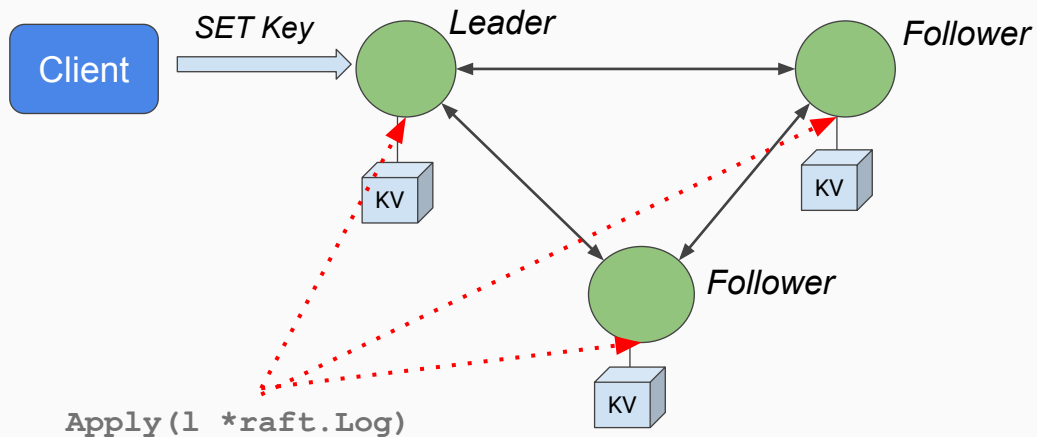
<https://www.philipotoole.com>

# Building a Raft-based Key-Value store

```
// Apply changes the KV store - called on each node only if consensus reached! 
func (s *KVService) Apply(*raft.Log) interface{} {
    m := map[string]string{}
    json.Unmarshal(log.Data, &m)
    for k, v := range m {
        s.kv[k] = v
    }
}

// NewKVService instantiates the distributed KV store
func NewKVService() *KVService {
    kv := &KVService{}
    kv.consensus = raft.NewRaft(raft.Network("localhost:10000"), (raft.FSM)(kv), ...)
    return kv
}
```

# Introducing *hraftd*



Philip O'Toole  
<https://www.philipotoole.com>

Now here **is** the system architecture of the distributed KV store. This system exists today – I built it, and it's called *hraftd*. It's available on GitHub. The client sends its requests to the Leader.

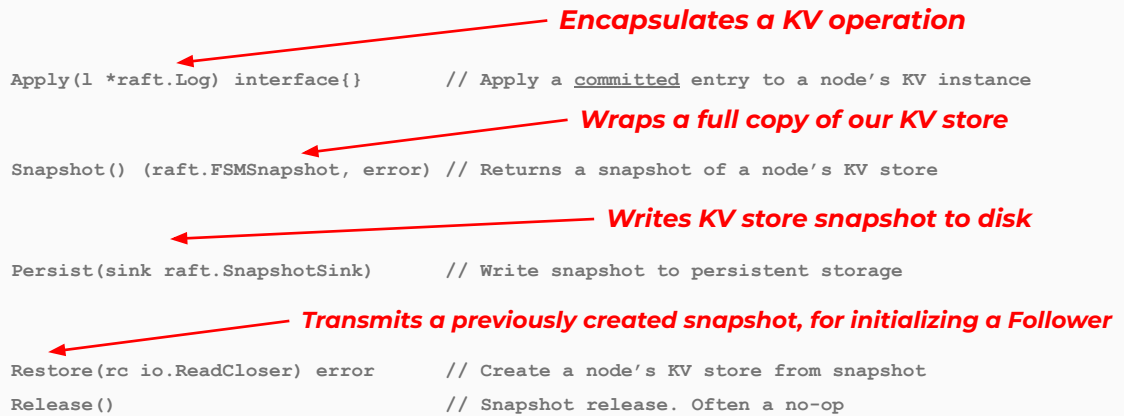
Let's look at the key parts:

- Each node runs the Raft consensus code
- Once consensus is reached for any operation sent to the Leader, actual KV updates are then – and only then – applied by each node to **its local KV store**. In coding terms the **Apply()** function you implement will be called.
- Which node is the Leader can change at anytime -- perhaps due to failure or a network fault.
- The Raft system will take care of this, ensuring a new Leader is elected within a few seconds.
- By the way this KV store is a CP system, in terms of the CAP Theorem. In the face of a network partition (the "P" in the CAP Theorem), one side of the cluster will continue to offer consistent reads and writes, whereas the other side of the cluster will be unavailable.
- In other words, the KV store provides consistency instead of availability.



# Hashicorp Raft programming interface

Integrating with the Raft consensus module involves implementing five key functions – the *Raft.FSM interface*.



```
Apply(1 *raft.Log) interface{}    // Apply a committed entry to a node's KV instance

Snapshot() (raft.FSMSnapshot, error) // Returns a snapshot of a node's KV store

Persist(sink raft.SnapshotSink)    // Write snapshot to persistent storage

Restore(rc io.ReadCloser) error     // Create a node's KV store from snapshot

Release()                          // Snapshot release. Often a no-op
```

**Encapsulates a KV operation**

**Wraps a full copy of our KV store**

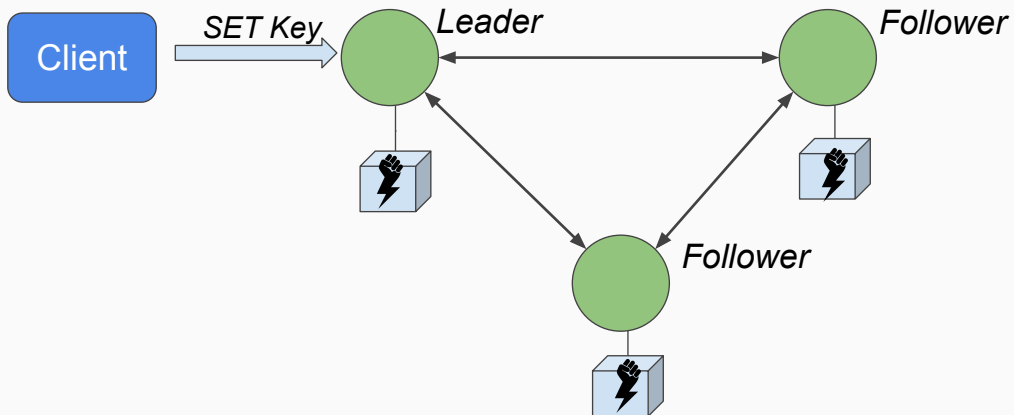
**Writes KV store snapshot to disk**

**Transmits a previously created snapshot, for initializing a Follower**

Philip O'Toole  
<https://www.philipotoole.com>

What is involved with integrating with Raft? Here is its API one must implement, which gives you an idea of the coding involved.

# Raft-based distributed BoltDB - system view



Philip O'Toole  
<https://www.philipotoole.com>

To learn this lesson, let's compare and contrast.

Now, let's see what changes if we decide to make a distributed version of BoltDB, the well-known KV store written by Ben Johnson, of Litestream fame. In this case the system design stays almost the same, but the Storage system to which we apply the events in the logs changes.

Let's see how using BoltDB would map to code changes.

# Integrating Raft with BoltDB



```
// KVService is a distributed BoltDB store which uses Raft
type KVService {
    db *boltDB.DB // Each Service has a BoltDB instance
    consensus *raft.Raft // Each Service still has a Raft module
}

func (s *KVService) SetKey(w http.ResponseWriter, r *http.Request) {
    buf := io.ReadAll(r.Body) // Read the request body
    future := s.consensus.Apply(buf, 5 * time.Second) // Call Raft!
    if future.Error() != nil {
        w.WriteHeader(http.StatusServiceUnavailable)
    }
}

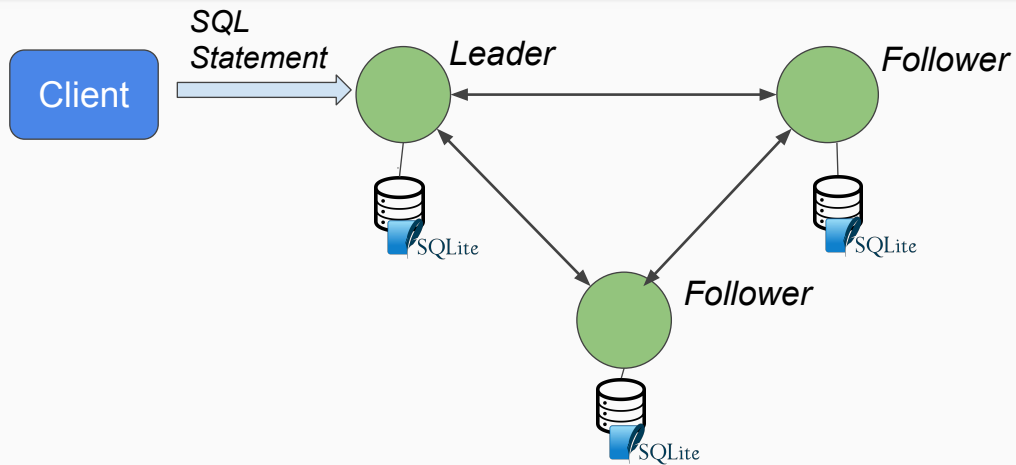
// Apply is the function that actually changes, everything else stays mostly the same!
func (s *KVService) Apply(*raft.Log) interface{} {
    m := map[string]string{}
    json.Unmarshal(log.Data, &m)
    b := boltDB.Tx.CreateBucketIfNotExists([]byte("TestBucket"))
    for k, v := range m {
        b.Put([]byte(k), []byte(v))
    }
}
```

Philip O'Toole

<https://www.philipotoole.com>

At a quick glance this code looks roughly the same. But look closer.

# Raft-based distributed SQLite - system view



Philip O'Toole  
<https://www.philipotoole.com>

Let's do the same thing with SQLite -- turning SQLite into a distributed version of itself. Again, the system diagram looks almost the same, but this time the log will contain SQL statements. This is basically how rqlite works.

One thing to notice is that this is statement-based replication. Understand why this may matter to your system.

And again, let's look at the code for this system.

# Integrating Raft with SQLite

```
curl -XPOST localhost:8080/db/execute -d '{"query": "CREATE TABLE foo (name TEXT)"}'
```

```
// KVService is a distributed SQLite database which uses Raft
type KVService {
    db *sql.DB // Each Service has a SQLite connection
    consensus *raft.Raft // Each Service still has a Raft module
}

func (s *KVService) Execute(w http.ResponseWriter, r *http.Request) {
    buf := io.ReadAll(r.Body) // Read the request body
    future := s.consensus.Apply(buf, 5 * time.Second) // Call Raft!
    if future.Error() != nil {
        w.WriteHeader(http.StatusServiceUnavailable)
    }
}

func (s *KVService) Apply(*raft.Log) interface{} { // Apply is the function that actually
    m := map[string]string{} // changes, everything else stays the same!
    json.Unmarshal(log.Data, &m)
    for _, v := range m {
        db.Exec(v)
    }
}
```



Philip O'Toole  
<https://www.philipotoole.com>

By now I hope you are seeing the pattern. Raft provides the distributed consensus system -- the log, the leader, the failure detection, and the leader election - and you decide what data goes into the log!

# Practicalities



Let's now talk about building and running these systems in the *real world*.

# Testing your system

- **Unit testing first and foremost**

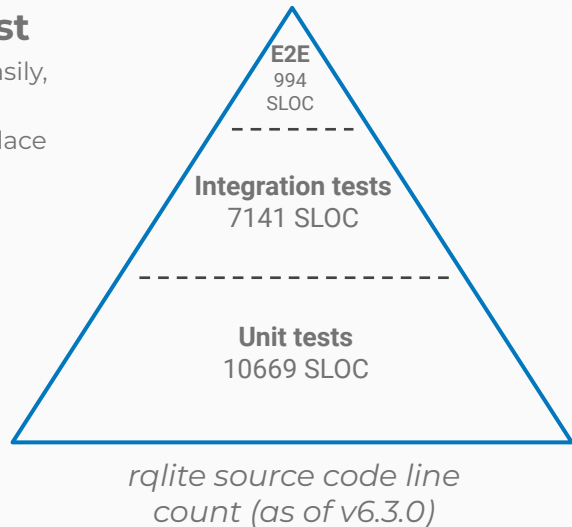
- If a component cannot be unit-tested easily, then its interfaces may be wrong
- Most storage-level testing should take place in unit tests

- **System-level testing should focus on Raft interactions**

- Integration testing
- Single and multi-node tests

- **Limited end-to-end testing**

- Ensures “happy path” functionality



Philip O'Toole  
<https://www.philipotoole.com>

- **Let's say you want to build something like this. How should you think about testing?**
- Quality testing -- and specifically the ability to test -- is very important for complicated and racy systems such as we've discussed. Let's take a look at the testing approach I have found effective. I have never needed to use anything but the standard Go unit test framework. *Keep your tests clear and easy to understand.*
- Unit testing is the fundamental test approach. When it comes to unit testing, if a component cannot be unit tested easily, it's a strong sign that the component's design or interface is wrong.
- If unit testing a component requires bringing in many other components, then it's not focused enough on one task.
- Most store-level testing takes place at the unit test level. This works quite well because these systems are a shared-nothing system when it comes to the storage layer, so there is no need to test beyond a single instance of the Go storage layer and whatever underlying technology you are storing your data in.
- Next comes system-level testing. System-level testing -- also known as integration testing -- focuses on the Raft consensus system and how it interacts with your storage layer.
- Functionality such as replication of commands to each node, various read-consistency behavior, and leader election, are tested at this layer.

- Finally there is **deliberately limited** end-to-end testing.
- End-to-end testing launches real distributed processes, runs real clusters, and ensures our system basically works.
- Tests should be added here sparingly because end-to-end testing can be time consuming, and difficult to debug.
- Tests are never added here if an equivalent test can be added to one of the other layers of testing. And if a test can only be performed in the end-to-end suite, it's important to determine if there's a flaw in your system's construction before proceeding.
- End-to-end testing is basically a smoke-check.
- This testing philosophy adheres quite closely to the [“Testing Pyramid” philosophy](#), where most testing is done as close as possible to the components themselves.
- Go has been effective when it comes to writing tests.
- For reference, rqlite v6.3.0 is about 36,000 lines of source code.



# Accessing and managing your cluster



**Finding the  
Leader**



**Adding a  
node**



**Reading  
data**

Philip O'Toole  
<https://www.philipotoole.com>

**Clients must be able to find the Leader** -- the existence of the Leader must be hidden from clients.

How can clients find the Leader? Well the first thing to remember is that every node knows which node is the Leader -- and where it is on the network. You could use sophisticated DNS, nodes could respond to clients telling those clients where the leader is, or Follower nodes can transparently forward requests to the Leader.

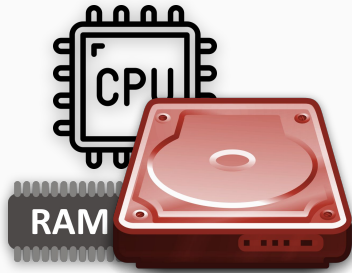
Transparent forwarding, by the way, is what rqlite does. But all of this starts to require substantial more coding that is present today in hraftd, the example Raft-based KV store up on GitHub.

Adding and removing nodes requires consensus.

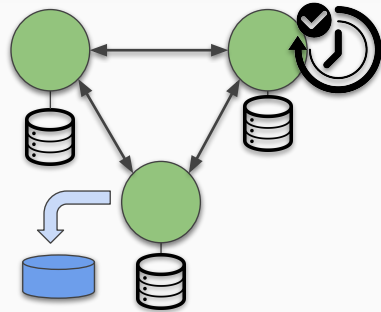
Kubernetes works great with Hashicorp Raft -based systems. It can help with finding the Leader as well as adding and removing nodes.

Remember bugs will happen! Be sure whatever system you build can easily backup its data set -- in our initial example, the KV store. Backup often, and backup the entire Raft log if you feel like it. It depends on your implementation.

# Monitoring your Raft-based System



**Hashicorp Raft is most sensitive to disk performance**



**Node start-up times  
Raft *Snapshot* times**

Philip O'Toole  
<https://www.philipotoole.com>

You should monitor all the usual suspects -- CPU, RAM, and disk. But disk is particularly important -- that is the first bottleneck you'll hit when pushing your system to the limit.

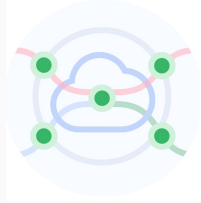
It's also particularly important to monitor node start-up times, and Raft snapshot times.

Of course, monitor the nodes themselves -- Hashicorp Raft has status reporting available on the Raft module, so you should expose this via HTTP (or whatever protocol makes sense for you). Make liberal use of the expvar module too.

# Should I choose Raft?



**Latency**



**Control plane  
vs. Data plane**



**Scaling vertically vs  
scaling horizontally**

Philip O'Toole  
<https://www.philipotoole.com>

There are some final other considerations you need to take into account when building with the Hashicorp library -- and Raft systems in general. Perhaps you're even wondering if Raft is the right approach for your project? One way to make this decision is to look at the potential costs and downsides of using Raft.

The first thing is latency. If you think about, your writes are bouncing between nodes until consensus is reached. This introduces latency. So you can expect your write performance to change. Batching can help a lot, but this latency is unavoidable.

Another choice you might want to make when building your system is whether you build a data-plane system, or a control plane system. I've just shown you a data plane system. Maybe if you just require config data in consensus you can use Raft.

Horizontal scaling vs vertical scaling - if you need a system that scales writes horizontally, Raft may not be you, not if you want to put your write traffic through Raft.

Will you nodes come and go? Do you care?

- Not necessarily a drop in – not without more work

Practical cluster sizes - odd, not even. hics on the right

- Clusters are most practical when 3, 5, or 7 nodes in size. Even numbers don't get you anything.
- A 3-node cluster has a quorum of 2, which means it can tolerate loss of a single node.
- A 4-node cluster has quorum of 3, which it too can only tolerate loss of a single node. You must go to a 5-node cluster to tolerate loss of 2 nodes.
- rqlite has been a successful example of software composition - modern distributed consensus system meets modern database. Neither SQLite nor Hashicorp's Raft implementation had to change.

# Summary

- We examined the core challenge of Distributed Systems – and how Raft helps
- Dived into the integration of Raft with 3 different data storage systems, using the Hashicorp Raft module
- Gained some practical insights for developing and testing your own Distributed System

*Philip O'Toole*  
<https://www.philipotoole.com>

# Thank you

*[www.philipotoole.com/gophercon2023](http://www.philipotoole.com/gophercon2023)*



*Philip O'Toole*  
<https://www.philipotoole.com>

Visit these web pages for more information:

- <https://raft.github.io>
- <http://thesecretlivesofdata.com/raft/>
- <https://github.com/hashicorp/raft>
- <https://github.com/otoolep/hraftd>
- <https://github.com/rqlite>
- <https://rqlite.io/docs/design/>
- <https://github.com/boltdb/bolt>
- <https://github.com/mattn/go-sqlite3>