At this point, our application is able to connect to the cluster and get information about any changes to the cluster topology. The receptionist in the remote actor system will accept a few different messages:

- `ClusterClient.Send`: This sends a message to a random node.
- `ClusterClient.SendToAll`: This sends a message to all actors in the cluster.
- `ClusterClient.Publish`: This sends a message to all actors subscribed to a topic.

We only need to send the message to a random worker; so, `Send` is fine for our case. We make the Send object, describing which actor the message is destined for (the "workers" router running on each node of the cluster) and wrapping our message in it:

```
newClusterClient.Send("/user/workers", articleToParse, false)
```

Finally, we use ask to send the message to the receptionist `ActorRef` to deliver, and to get the result back (shown calling ask method, but in Scala we can also use the '?' operator):

```
Patterns.ask(receptionist, msg, timeout);
```

We take the future that ask gives us and wait for the result—again, this is only for the sake of an example. You never want to block threads by waiting in your code.

If a node becomes unavailable when you try to send it a message, your request will time out and fail. It is your responsibility to handle timeout and retry semantics, or otherwise handle failure cases.

That's all that it takes to use Akka to build distributed workers. Now, there are a lot of things we would want to do to further improve this system. If there are potentially lots of messages, we would likely want to put the messages somewhere other than in the mailbox in memory. Likely, some sort of durable queue or database would be used instead of ephemeral memory. Though, for real-time processing, this isn't a bad start. For a client, we would likely want to build timeout and retry mechanics as well to ensure that we always get the work done that we need done.

Building a Distributed Key Value Store

We've looked at everything that it takes to build on top of Akka cluster and we looked at a small example of a stateless cluster of workers. If state is involved, the problem becomes incredibly difficult to get right.

In the next section, we'll look at the watchouts, tools, and techniques for handling distributed systems that contain state such as a key-value store.

Disclaimer – Distributed Systems are Hard

Before we go on to look at how to build a distributed key-value store, I want to give you a word of warning.

It's not terribly difficult to build distributed systems that appear to work perfectly fine. You might get confident and feel that it's not all that hard after all. You'll tout yourself soon as an expert Distributed Systems Person. But stay humble—in reality, things fail, networks partition, and services become unavailable—and gracefully handling those scenarios without data loss or corruption is an incredibly difficult problem. Perhaps, even an unsolvable one with our network technology today.

How applications respond in those error cases can be more important than their primary functionality because failure is common and, in fact, is inevitable as you scale up.

Remember that the network is not reliable. In a cluster of 1,000 nodes running in AWS, it's very likely that there will be some service interruption somewhere in your system at any given time.

So, how do all of these new-fangled `NoSql` data-stores actually stack up in their claims of availability and partition tolerance then? There is an interesting series of articles called *Jepsen* or *Call Me Maybe* on `aphyr.com` that tests several distributed system's documented claims about resiliency—you might be surprised by how your favorite technology fairs in some of the tests run in the articles.

This book will not be able to explain all of the techniques or examine all of the solutions to these problems, so all that I can do as the author is to warn you that it's a real can of worms and nobody has done a perfect job at solving the distribution problem yet.

Designing the Cluster

We are going to build a very simple three-node data-store to start demonstrating the techniques that can be used in distributed systems. We are going to look at a couple different designs so that you understand the techniques that are used and what the problems are that distributed systems try to solve.

First, let's begin talking about our cluster and how the client interacts with it. Ideally, we need our multi-node key-value store to offer a few features:

- Mechanisms for replicating data to gracefully handle a node partition
- Mechanism for giving a consistent view of replicated data—to give the most recent update when a client requests it (consistency)
- Mechanism for linear scalability—20 nodes can handle twice as much throughput as 10 nodes

In terms of design goals in meeting those objectives, we would like to have a database that is highly available, partition tolerant, and able to give a consistent view of the most recent data. Achieving all three perfectly is not likely possible with our technology today, but we can take measures to try. We're lucky that a lot of very smart people have been working on these problems; thus, we have access to their research and publications.

Basic Key-Value Store Design

We've already looked at how a single node can store an object in a map with a key associated with it. Using a `HashMap` gives approximate constant time lookups, so it's a very efficient choice for storing data in memory.

An actor will accept `Get` messages with a key, and Put messages with a key and a value—exactly as demonstrated in *Chapter 2*, *Actors and Concurrency*. To recap, our node will have an actor that looks like the following:

```
//Java
    private AkkademyDb(){
         receive(ReceiveBuilder.
 match(SetRequest.class, message -> {
                              log.info("Received Set request: {}",
message);
 map.put(message.key, message.value);
                              sender().tell(new Status.Success(message.
key), self());
                          }).
                          match(GetRequest.class, message -> {
                              log.info("Received Get request: {}",
message);
                              Object value = map.get(message.key);
                              Object response = (value != null)
                                      ? value
                                      : new Status.Failure(new
KeyNotFoundException(message.key));
                              sender().tell(response, self());
                          }).
```

```
 matchAny(o ->
                                    sender().tell(new Status.Failure(new
ClassNotFoundException()), self())
                                ).build()
          );
      }

 //Scala
   override def receive = {
     case SetRequest(key, value) =>
       log.info("received SetRequest - key: {} value: {}", key, value)
 map.put(key, value)
       sender() ! Status.Success
     case GetRequest(key) =>
       log.info("received GetRequest - key: {}", key)
 val response: Option[Object] = map.get(key)
       response match{
         case Some(x) => sender() ! x
         case None => sender() ! Status.Failure(new
KeyNotFoundException(key))
       }
     case o =>Status.Failure(new ClassNotFoundException)
   }
```

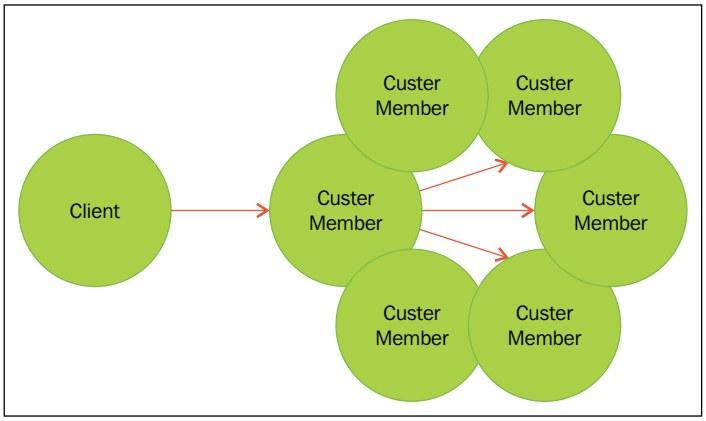
I would recommend that you add other messages for common use cases if you're going to attempt to implement this. Semantics like `SetIfNotExist`, which sends back a failure if the node is already there, is useful for handling concurrency consistently—getting and then setting the value if it does not exist is not at all consistent enough for a distributed workflow. Redis API docs are a good resource to look at if you want to see the types of messages and datatypes that are useful for a key-value store to handle with multiple clients—it's a simple and readable document that will go a long way to ramp you up in the problem space.

Coordinating Node

Now that we have a picture of how we'll store data in a node, we need to look a bit higher level at how we want messages to be handled between a client using the data-store and the cluster.

We have a client example that sends a message to a random node and that's actually quite a good start for most distributed stores because very often such systems will implement the concept of a coordinating node that can be any node in the cluster that will handle talking to other nodes in the cluster to handle the request.

It's probably unclear at this point exactly why we would do this, but let's imagine a coordinating node that needs to talk to three other nodes to get a definitive answer on what a value is.



If we implement this logic, a client can send the request to any node in the cluster, and that node becomes the coordinator for the request. That node then goes to the other nodes and requests the data that it needs to make a decision on what the value is that is being retrieved. So, we can continue to use the random delivery mechanism if we implement the coordination logic on the server instead of the client. Because we're not sure where the client is located, it's safer to move this responsibility into the cluster itself—we can be more confident that the nodes we need to talk to have lesser leaps across the network in the majority of circumstances. This is how Cassandra handles requests for example.

We'll look at two models: one for storing a data set across multiple nodes and one for replicating data across multiple nodes.

Sharding for Linear Scalability

The first problem we will look to solve is how to assign a piece of a problem domain to different nodes in a cluster. In the case of storing data in a key-value store, it's easy to see how you might take slices of data (determined by the key) and assign them to different nodes. In Cassandra, a key used to determine which node data goes to is called a partition key.

For our use case of a very simple key-value store, we can take a key and hash it, and then execute modulus on the hash to get an integer value. Let's assume we have three nodes that we want to store data on. A request goes to a node in the cluster to store a key-value pair—for example, `foo` as the key and `bar` as the value. The coordinating node will execute `hashCode()` on the key:

```
"foo".hashCode()
 (101574)
```

Then, it will call the modulus operator on the `hashcode`, giving a result of `0`, `1`, or `2`:

```
101574 % 3
 (0)
```

Now, if we have three nodes that we want to store data on, then we know if we want to set or get a value with the key `foo` that it goes on the first node. This approach of sharding data is used in many distributed data-stores today.

Redundant Nodes

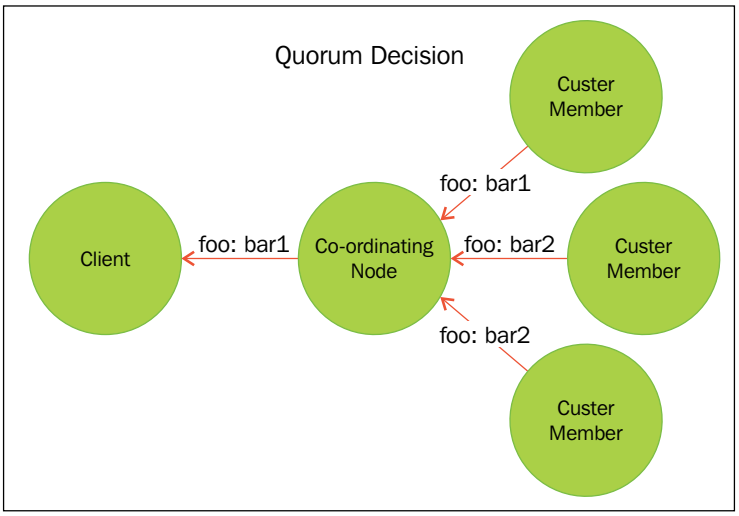
To have high availability and partition tolerance, we have to tolerate a node disappearing from the cluster for both reads and writes. If one node disappears, this should be a non-critical failure and the data-store should continue to operate to meet our goals.

To accomplish this, we can send all writes to three nodes and hope that the majority of them respond with an acknowledgment. If only two respond with an ok, we might be able to handle that on future reads. Now, all the intricacies of this mechanism—and specifically how to determine the ordering of events—cannot be covered here, but you can look at lamport clocks or vector clocks if you're motivated to get this right. We'll look at the mechanics at a high level here using simpler mechanics that are easier to comprehend.

Let's say a client wants to write a value for key `foo` of value `bar`, we want to persist this to three nodes. We'll try to write the value to three nodes from the coordinating node.

We can reference the previous diagram—a client will make a request to write and it goes to three nodes in the cluster. To have a successful write, we might agree that at least two of the nodes need to acknowledge the write, or maybe all three do. Adjusting these values tune how it responds to partitions and node failures.

Now, if we want to retrieve the value, then we can request the data from any of the three nodes as a starting point, but what if one node goes down and misses some writes? In this case, we can request the data from all three nodes, and require the same value from at least two nodes.



Now, we have some way of determining what the majority of nodes think the value is—we say that two nodes must have the same value to determine which it should be.

But how do we know the order of values? Was `bar1` or `bar2` written first? Which is more recent? We actually don't have any way of determining the ordering of messages, which it turns out is quite a problem. What happens if one node gets writes in a different order than another node? Cassandra has another read request type called a read repair request, which compares the data stored for a key on all of the replica nodes and tries to propagate the most recent write to the replicas.

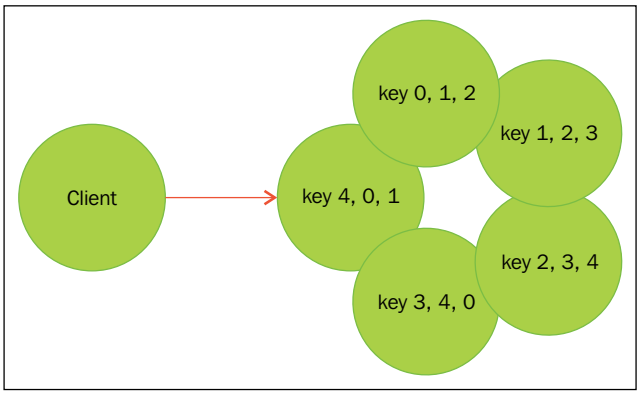
The actual ordering of events is an interesting problem. You could start by using a timestamp in the record to determine which records are most recent, but we can't guarantee that all machines have the same clock or that events are occurring infrequently enough to trust the timestamp.

There are a couple of papers and algorithms you can look at that describe the problem and some solutions. Some of the items you may want to look at are Vector Clocks (Used by many distributed technologies such as Akka and Cassandra) and Dotted Version Vectors (used by Basho's Riak in recent versions).

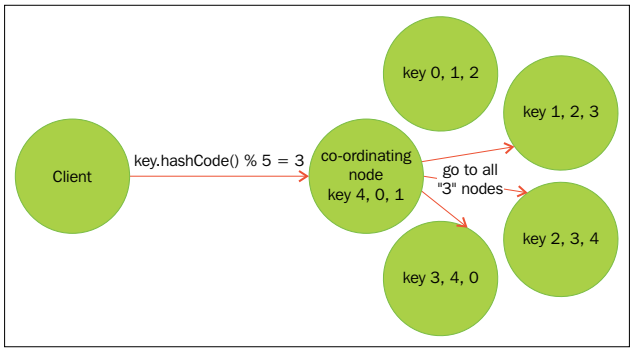
- The Dynamo paper from Amazon describes the use of Vector Clocks in Amazon's Dynamo distributed key-value store—`http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf`
- Lamport wrote a paper on the problem of ordering in distributed systems in 1978 that is worth a look as well—`http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf`

Combining Sharding and Replication

Once we have approaches in place for sharding and replication, we will be able to combine the two together by first sharding data into a ring and then replicating the data to the neighboring nodes. Let's say we have five partition keys, and we want to replicate the data across three nodes; with five nodes, we end up with a topology that looks like the following, with the numbers representing the hash of the partition key as demonstrated:



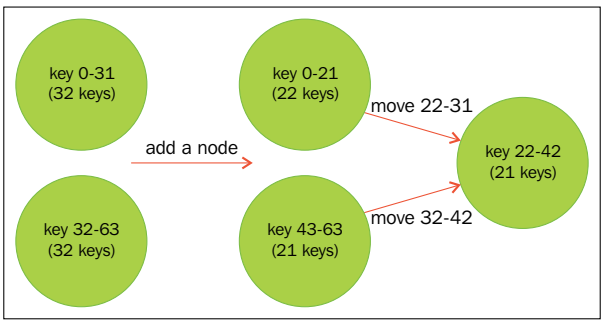
The request goes to a random node that becomes the coordinating node. That node would then calculate key partition key and go to the three nodes that have the data set. The retrieval and handing of the data at that point is no different than what we have looked at.



Pre-Sharding And Redistributing Keys to New Nodes

What happens if you want to be able to add new nodes? For example, what if you want to keep your replication factor of `3` and add two new nodes? If our partition key is only `0-4`, then we can't move the data across new nodes.

The trick to this problem is to not use five partition keys but to use a much larger number. Cassandra has a concept of a vnode—a virtual node—whereby a cluster has a larger number of shards than nodes out of the gate. If you start by sharding your database into 64 or 128, then you can add new nodes and move a portion of the partition keys to new nodes as they come into the cluster.



Then the coordinating node can talk to the new node, understanding it has been assigned a new set of partition keys. Note that moving data around to a new node is a one-time operation and likely requires you to stop all operations on all nodes until the re-distribution of partition keys is complete. If your partition is under heavy write through the operation, it might be difficult to ensure no data loss through the operation otherwise.

It will be simpler to assign a fixed size to the cluster until you can eventually get to these features. Akka Cluster has mechanisms to only start a cluster once it reaches a certain size. If you want to try building your own distributed key-value store, you should assume it has a fixed size and a known number of partition keys, and add pre-sharding and re-distributing keys as features later because they are non-trivial problems to solve.

Addressing Remote Actors

We've focused heavily on implementation details, but to round out this chapter, we should have a quick look at Actor Addressing again and how to get references to remote actors.

If we have an `ActorRef`, calling `actorRef.path()` will give us the actors URL—akka://ActorSystem/user/actor.

There are two parts to the path:

- **authority**: `akka://ActorSystem`
- **path**: `/user/actor`

Authority can be either local (`akka://ActorSystem`) or remote (`akka.tcp://ActorSystem@127.0.0.7`). The path will be the same for both local and remote actors `(/user/actor` for example).

An `ActorRef` will have an `ActorPath` with a fragment `#123456`, which is what is called the actor's UID. An `ActorPath` will not—it will just have the path.

We looked at `ActorSystem.actorSelection` earlier, which will let us look up an Actor with any path and send it messages:

```
ActorSelectionactorSelection = actorSystem.actorSelection("akka.tcp://
ActorSystem@127.0.0.7/user/my-actor");
valactorSelection = actorSystem.actorSelection("akka.tcp://
ActorSystem@127.0.0.7/user/my-actor")
```

That gives us an actor selection, which we can send messages to. Although, the actor selection does not assume that an actor exists. If we send messages to an `ActorSelection` and the actor does not exist, the message will disappear. Thus, we can look up an actor and get an `ActorRef` for a remote actor.

Using akka.actor.Identify to Find a Remote Actor

All actors by default accept a message—`akka.actor.Identify`. We can create a new Identify message and send it to an actor to get an `ActorRef`:

```
Identify msg = new Identify(messageId)
 Future<ActorIdentity> identity = (Future<ActorIdentity>) Patterns.
ask(actor, msg, timeout);
valmsg = Identify(messageId)
 val identity: Future[ActorIdentity] = (sentinelClient ? msg).
mapTo[ActorIdentity]
```

We will receive a response—`ActorIdentity(messageId, Some(actorRef))`, or `ActorIdentity((path, client), None)`—showing that an actor is either present or absent. This gives us a mechanism for determining if a remote actor exists and obtaining an `ActorRef` for it.

Homework

This chapter is a blend of theory and technique. We covered how to use Akka cluster to run work on and then how to use it to distribute datasets partitioned by key. The problems are too big to cover in an introductory chapter to Cluster, but if you try to approach some of the problems presented in this chapter, you should start to get a good handle on how people solve some of these problems today.

- Build your own worker queue using Cluster.
- Build a distributed key-value store with replication.
- Try to solve the linearization problem—how can you determine ordering? How can you "repair" nodes that fail and then recover?
- Build a distributed key-value store with sharding.
- Can you combine sharding and replication with minimal changes to the code you have now as described?

- In the sharded model, can you develop a way to redistribute data to new nodes? Is it harder to do this with replication? About the same difficulty?
- If you build a project, you should open source it and share your discoveries and learning!

Summary

In this chapter, we covered the basics of how you might design some different distributed systems. This gives you a few models that you can use for various problems—don't simply think of sharding/partitioning when dealing with data—you can use similar techniques for many real-time system problems to be able to scale out.

If you'd like to learn more about the techniques introduced here, you should both continue your learning and try to build the solutions to these problems yourself. In my humble opinion, the best way to learn is to teach and share, so try to start a distributed computing club or otherwise get some presentations together for your peers on how these technologies work—organizing your thoughts will help you get into the details and find your own way in discovering how we're moving to solve these types of distributed computing problems today.

In the next chapter, we'll discuss what happens in mailboxes when our actors get put under heavy load and look at how we can adjust mailbox configuration to adapt to those conditions.