Ringpop Documentation

Release 0.1.0

Uber Technologies, Inc.

May 19, 2016

Contents

Ringpop is a library that maintains a consistent hash ring atop a membership protocol. It can be used by applications to arbitrarily shard data in a scalable and fault-tolerant manner.

To find out more, head to the Getting Started section below.

CHAPTER 1

Getting Started

If you're looking for more information about what Ringpop is all about and how it might be able to help, you've come to the right place. Let's begin by digging deeper into Ringpop and why you'd want to use it.

1.1 What is Ringpop?

As we've stated in the introduction, Ringpop is a library that maintains a consistent hash ring and can be used to arbitrarily shard the data in your application in a way that's adaptable to capacity changes and resilient to failure.

Ringpop is best described by introducing its 3 core features: a membership protocol, a consistent hash ring and request forwarding. You can find more information about each of these in the Architecture and Design section. For the eager reader, its membership protocol provides a distributed application, whose instances were once completely unaware of one another, with the ability to discover one another, self-organize and cooperate. The instances communicate over a TCP backchannel and pass information between them in an infection-style manner. Enough information is shared to allow these instances to come to an agreement, or converge, on whom the participating instances, or members, are of the distributed application.

With a consistent membership view, Ringpop arranges the members along a consistent hash ring, divides up the integer keyspace into partitions and assigns ownership of the partitions to the individual instances of your application. It then projects a keyspace of your choosing, say the ID range of the objects in your application, onto that same ring and resolves an owner for each ID. In the face of failure, the underlying membership protocol is resilient and automatically reassigns ownership, also known as rebalancing, to the surviving instances.

Requests that your application serves, be it ones that create new objects, update or read or delete existing ones, may be sent to any instance. Each Ringpop instance is equipped to route the request to the correct owner should the shard key, for example the object's ID, resolve to an instance that is not the one that received the original request.

By maintaining a consistent hash ring based upon the information that is collected by its membership protocol and offering request forwarding as a routing convenience, Ringpop provides some very powerful and cool building blocks. What you as an application developer choose to do with these building blocks is entirely up to you. It may lead you to build an ultra scalable and highly available database, an actor model system, systems that are capable of electing a leader and delegating work to it, a request coalescing proxy, general purpose application-layer caches and much more. If you find some cool new ways to leverage Ringpop, let us know!

1.2 Why use Ringpop?

Ringpop is first and foremost an application developer's library. It is not an external system nor a shared infrastructure resource used by many applications. It allows your application to remain autonomous and not beholden to a dependency for its ability to scale and remain available. Ringpop promotes scalability and fault tolerance as an application

layer concern while keeping complexity and operational overhead to a minimum. Application developers have to be aware how their data is distributed, what makes that data available and how their application degrades in the face of failure. When using Ringpop you are sacrificing consistency for higher availability and one must take into consideration how even higher degrees of availability are achieved through techniques like replication and application-side conflict resolution. We've found that taking ownership of your application to such a degree is not only empowering, but a very sustainable and scalable practice.

Clients of your application can remain completely unaware that Ringpop is being used. They neither have to understand the underlying partitioning scheme nor who the correct recipient is for a request. There is no special technology that need exist between client and server. You may use load balancers, proxies, overlay networks, etc without fear of Ringpop incompatibilities.

Ringpop offers a rich administration API to inspect and control your cooperative application and easy to use tooling to help you understand the behavior of Ringpop and your application.

Lastly, Ringpop's sharding capabilities are just one application of what we see as a collection of composable distributed systems programming building blocks. We typically want our applications to be more cooperative when something needs to be made more efficient at a large scale or a resource in your system has to have a particular home or rendezvous point. We discover new ways to use Ringpop all the time and you'll likely run into a few interesting ways too.

1.3 Installation

```
npm install ringpop
```

CHAPTER 2

Running Ringpop

Before we get to programming against Ringpop, let's just run it by itself and see what happens. There are several ways to accomplish this and they are documented below. There's nothing too fancy going on when Ringpop runs by itself. To reap the full potential of it, you'll need to embed it into your application and start divvying incoming requests based on the sharding key of your choice. No matter, we're here, in this section, to get a look at what happens in Ringpop at steady-state, how its membership protocol behaves and how to launch a standalone version of it from the command-line. Let's get to it!

2.1 Running with tick-cluster

`tick-cluster` is a utility located in the `scripts/` directory of the Ringpop repo that allows you to quickly spin up a Ringpop cluster of arbitrary size and test basic failure modes: suspending, killing and respawning nodes.

To use `tick-cluster`, first clone the repo and install Ringpop's dependencies:

```
$ git clone git@github.com:uber/ringpop.git
$ npm install
```

Then run `tick-cluster`:

```
$ ./scripts/tick-cluster.js [-n size-of-cluster] [-i interpreter-that-runs-program] <ringpop-program>
```

`tick-cluster` will spawn a child process for each node in the cluster. They will bootstrap themselves using an auto-generated `hosts.json` bootstrap file and converge on a single membership list within seconds. Commands can be issued against the cluster while `tick-cluster` runs. Press `h` or `?` to see which commands are available.

Whenever it is specified, the program is run by an interpreter, otherwise the program should be a binary. The cluster size defaults to 5.

Here's a sample of the output you may see after launching a 7-node cluster with `tick-cluster`:

```
$ ./scripts/tick-cluster.js -n 7 -i node ./main.js
[init] 11:11:52.805 tick-cluster started d: debug flags, g: gossip, j: join, k: kill, K: revive all,
[cluster] 11:11:52.807 using 10.80.135.224 to listen
[init] 11:11:52.818 started 7 procs: 76365, 76366, 76367, 76368, 76369, 76370, 76371
```

2.2 Running from the command-line

Content coming soon...

2.3 Administration

Content coming soon...

2.4 Configuration

Content coming soon...

2.5 Deploying

Content coming soon...

2.6 Monitoring

Ringpop emits stats by making use of the dependency it has on a Statsd- compatible client. It emits all stats with a prefix that includes its identity in the stat path, e.g. `ringpop.10_30_8_26_20600.*`; the dots and colon are replaced by underscores. The table below lists all stats that Ringpop emits:

|Node.js Path|Description|Type |—-|—-|—- |changes.apply|Number of changes applied per membership update|gauge |changes.disseminate|Number of changes disseminated per request/response|gauge |checksum|Value of membership checksum|gauge |compute-checksum|Time required to compute membership checksum|timer |damp-req.recv|Damp-req request received|count |damp-req.send|Damp-req request sent|count |damper.damp-req.damped|Damp-req resulted in members being damped|count |damper.damp-req.error|Damp-req resulted in an error|count |damper.damp-req.inconclusive|Damp-req results were inconclusive|count |damper.flapper.added|Flap damping detected a flappy node|count |damper.flapper.removed|Flap damping removed a flappy node|count |damper.flappers|Number of current flappers|gauge |dissemination.bump-bypass|Number of times piggyback count is preserved after failed ping or ping-req|count |filtered-change|A change to be disseminated was deduped|count |full-sync|Number of full syncs transmitted|count |heal.triggered.discover_provider|Number of times the partition healing is initiated using the discover provider. Note: this stat will be emitted even if there is no faulty or unknown target found; the actual number of heal attempts can be measured using `heal.attempt`.|count |heal.attempt|Number of times a heal opeartion is performed to a target-node|count |join|Time required to complete join process successfully|timer |join.complete|Join process completed successfully|count |join.failed.destroyed|Join process failed because Ringpop had been destroyed|count |join.failed.err|Join process failed because of an error|count |join.recv|Join request received|count |join.retries|Number of retries required by join process|gauge |join.succeeded|Join process succeeded|count |lookup|Time required to perform a ring lookup|timer |make-alive|A member was declared alive|count |make-damped|A member was declared damped|count |make-faulty|A member was declared faulty|count |make-leave|A member was declared leave|count |make-suspect|A member was declared suspect|count |max-piggyback|Value of the max piggyback factor|gauge |membership-set.alive|A member was initialized in the alive state|count |membership-set.faulty|A member was initialized in the faulty state|count |membership-set.leave|A member was initialized in the leave state|count |membership-set.suspect|A member was initialized in the suspect state|count |membership-set.unknown|A member was initialized in an unknown state|count |membership-update.alive|A member was updated to be alive|count |membership-update.faulty|A member was updated to be faulty|count |membership-update.leave|A member was updated in the leave state|count |membership-update.suspect|A member was updated to be suspect|count |membership-update.unknown|A member was updated in the unknown state|count |membership.checksum-computed|Membership checksum was computed|count |not-ready.ping|Ping received before Ringpop was ready|count |not-ready.ping-req|Ping-req received before Ringpop was ready|count |num-members|Number of members in the membership|gauge |ping|Ping response time|timer |ping-req|Ping-req response time|timer |ping-req-ping|Indirect ping sent|timer |ping-req.other-members|Number of members selected for ping-req fanout|timer |ping-req.recv|Ping-req request received|count |ping-req.send|Ping-req request sent|count |ping.recv|Ping

request received|count |ping.send|Ping request sent|count |protocol.damp-req|Damp-req response time|timer |protocol.delay|How often gossip protocol is expected to tick|timer |protocol.frequency|How often gossip protocol actually ticks|timer |refuted-update|A member refuted an update for itself|count |requestProxy.checksumsDiffer|Checksums differed when a forwarded request was received|count |requestProxy.egress|Request was forwarded|count |requestProxy.inflight|Number of inflight forwarded requests|gauge |requestProxy.ingress|Forward request was received|count |requestProxy.miscount.decrement|Number of inflight requests were miscounted after decrement|count |requestProxy.miscount.increment|Number of inflight requests were miscounted after increment|count |requestProxy.refused.eventloop|Request was refused due to event loop lag|count |requestProxy.refused.inflight|Request was refused due to number of inflight requests|count |requestProxy.retry.aborted|Forwarded request retry was aborted|count |requestProxy.retry.attempted|Forwarded request retry was attempted|count |requestProxy.retry.failed|Forwarded request failed after retries|count |requestProxy.retry.reroute.local|Forwarded request retry was rerouted to local node|count |requestProxy.retry.reroute.remote|Forwarded request retry was rerouted to remote node|count |requestProxy.retry.succeeded|Forwarded request succeeded after retries|count |requestProxy.send.error|Forwarded request failed|count |requestProxy.send.success|Forwarded request was successful|count |ring.change|Hash ring keyspace changed|gauge |ring.checksum-computed|Hash ring checksum was computed|count |ring.server-added|Node (and its points) added to hash ring|count |ring.server-removed|Node (and its points) removed from hash ring|count |updates|Number of membership updates applied|timer

2.7 Benchmarks

Content coming soon...

2.8 Troubleshooting

Content coming soon...

CHAPTER 3

Programming Ringpop

You may decide that Ringpop is right for your application and want to know how to program against it. Below you'll find information about how to use Ringpop within your application and the API Ringpop exposes to the application developer.

3.1 Code Walkthrough

The first thing you'll want is a handle on a Ringpop instance. You'll first need an instance of TChannel, the underlying transport for Ringpop:

Node.js

```
var TChannel = require('TChannel');

var tchannel = new TChannel();
var subChannel = tchannel.makeSubChannel({
    serviceName: 'ringpop',
    trace: false
});
```

Then decide on a listening address for TChannel. We'll leave that exercise for the reader. For the purposes of this exercise, let's assume we're using:

Node.js

```
var host = '172.18.27.228';
var port = 3000;
```

You're almost ready for Ringpop. Before we get to it, we'll need a list of addresses that act as the seed for Ringpop to join a cluster of other nodes. Let's assume that there are other Ringpop nodes that are or will be available:

Node.js

```
var bootstrapNodes = ['172.18.27.228:3000', '172.18.27.228:3001',
    '172.18.27.228:3002'];
```

We're there! Instantiate Ringpop:

Node.js

```
var ringpop = new Ringpop({
    app: 'yourapp',
    hostPort: host + ':' + port,
    channel: subChannel
```

```
});
ringpop.setupChannel();
ringpop.channel.once('listening', onListening);
ringpop.channel.listen(port, host);

function onListening() {
    ringpop.bootstrap(bootstrapNodes, onBootstrap);
}

function onBootstrap(err) {
    if (err) {
        // Fatal error
        return;
    }

    // Start listening for application traffic
}
```

When TChannel starts listening for connections, Ringpop is ready to be bootstrapped. Bootstrapping consists of having Ringpop send out a join request to a number of random hosts selected from `bootstrapNodes`. Your application is ready to serve traffic when Ringpop successfully joins a cluster.

As requests arrive, you'll want to lookup a request's key against the ring. If the key hashes to the address of Ringpop (`hostPort`), your request may be handled locally, otherwise it'll need to be forwarded to the correct Ringpop. The typical pattern you'll see looks similar to:

Node.js

```
var destination = ringpop.lookup(key);

if (destination === ringpop.whoami()) {
    // Handle the request
} else {
    // Forward the request
}
```

This pattern has been codified in Ringpop's `handleOrProxy` function as a convenience and can be used to forward HTTP traffic over TChannel. If the request should not be handled locally, `handleOrProxy` will return `false`:

Node.js

```
if (ringpop.handleOrProxy(key, req, res, opts)) {
    // Handle the request
}
```

That's really all there is to using Ringpop within your application. For a deeper dive into all of the other bells and whistles Ringpop has to offer we refer you to the API section below and the Running Ringpop page.

3.2 API

3.2.1 `Ringpop(opts)`

Creates an instance of Ringpop.

- `app` - The title of your application. It is used to protect your application's ring from cross-pollinating with another application's ring. It is a required property.
- `channel` - An instance of TChannel. It is a required property.

- `hostPort` - The address of your Ringpop. This is used as the node's identity in the membership protocol and the ring. It is a required property.

NOTE: There are many other options one can specify in the Ringpop constructor. They are not yet documented.

3.2.2 `bootstrap(bootstrapFileOrList, callback)`

Bootstraps Ringpop; joins the cluster.

- `bootstrapFileOrList` - The path of a bootstrap file on disk. Its contents are expected to be a JSON array of Ringpop addresses. Ringpop will select a number of random nodes from this list to which join requests will be sent. Alternatively, this argument can be a Javascript array of the same addresses.
- `callback(err)` - A callback.

3.2.3 `handleOrProxy(key, req, res, opts)`

Acts as a convenience for the "handle or forward" pattern.

- `key` - An arbitrary key, typically a UUID. Its hash code is computed and its position along the ring is found. Its owner is the closest node whose hash code is closest (in a clock-wise direction)
- `req` - Takes the shape of a Node.js http.ClientRequest
- `res` - Takes the shape of a Node.js http.ServerResponse
- `opts` - Valid options are listed below.

opts

- `bodyLimit` - The maximum size of the allowable request body. Default is 1MB.
- `endpoint` - The TChannel endpoint to which the request will be forwarded. The default is /proxy/req. Typically, this should be left untouched.
- `maxRetries` - Maximum number of retries to attempt in the event that a forwarded request fails.
- `retrySchedule` - An array of delay multiples applied in between each retry. Default is `[0, 1, 3.5]`. These multiples are applied against a 1000ms delay.
- `skipLookupOnRetry` - A boolean flag to specify whether a request should be rerouted if the ring changes in between retries.
- `timeout` - The amount of time, in milliseconds, to allow for the forwarded request to complete.

3.2.4 `lookup(key)`

Looks up a key against the ring; returns the address of the Ringpop that owns the key.

- `key` - See the description of `key` above.

3.2.5 `lookupN(key, n)`

Looks up a key against the ring; returns the addresses of `n` distinct Ringpop's that own the key; useful for replication purposes. If `n` are not found, fewer addresses may be returned.

- `key` - See the description of `key` above.

- `n` - The number of secondary owners aka a "preference list".

3.2.6 `whoami()`

Returns the address of Ringpop

3.2.7 Events

Content coming soon...

3.3 An Example Express App

Let's see all of this come together in an example web application that you can run and curl yourself. This is a 3-node Ringpop cluster each with its own HTTP front-end capable of 'handling' and forwarding HTTP requests in less than 100 lines of code.

To run:

1. Paste this code into a file named `example.js`
2. `npm install ringpop@10.8.0 tchannel@2.8.0 express`
3. `node example.js`
4. curl to your heart's content: `curl 127.0.0.1:6000/objects/abc`

Node.js

```
var express = require('express');
var Ringpop = require('ringpop');
var TChannel = require('TChannel');

var host = '127.0.0.1'; // not recommended for production
var ports = [3000, 3001, 3002];
var bootstrapNodes = ['127.0.0.1:3000', '127.0.0.1:3001',
    '127.0.0.1:3002'];

var cluster = [];

// Create Ringpop instances
ports.forEach(function each(port) {
    var tchannel = new TChannel();
    var subChannel = tchannel.makeSubChannel({
        serviceName: 'ringpop',
        trace: false
    });

    cluster.push(new Ringpop({
        app: 'yourapp',
        hostPort: host + ':' + port,
        channel: subChannel
    }));
});

// Bootstrap cluster
cluster.forEach(function each(ringpop, index) {
```

```
    ringpop.setupChannel();
    ringpop.channel.listen(ports[index], host, function onListen() {
        console.log('TChannel is listening on ' + ports[index]);
        ringpop.bootstrap(bootstrapNodes,
            bootstrapCallback(ringpop, index));

        // This is how you wire up a handler for forwarded requests
        ringpop.on('request', forwardedCallback());
    });
});

// After successfully bootstrapping, create the HTTP server.
var bootstrapsLeft = bootstrapNodes.length;
function bootstrapCallback(ringpop, i) {
    return function onBootstrap(err) {
        if (err) {
            console.log('Error: Could not bootstrap ' + ringpop.whoami());
            process.exit(1);
        }

        console.log('Ringpop ' + ringpop.whoami() + ' has bootstrapped!');
        bootstrapsLeft--;

        if (bootstrapsLeft === 0) {
            console.log('Ringpop cluster is ready!');
            createHttpServers();
        }
    };
}

// In this example, forwarded requests are immediately ended. Fill in with
// your own application logic.
function forwardedCallback() {
    return function onRequest(req, res) {
        res.end();
    }
}

// These HTTP servers will act as the front-end
// for the Ringpop cluster.
function createHttpServers() {
    cluster.forEach(function each(ringpop, index) {
        var http = express();

        // Define a single HTTP endpoint that 'handles' or forwards
        http.get('/objects/:id', function onReq(req, res) {
            var key = req.params.id;
            if (ringpop.handleOrProxy(key, req, res)) {
                console.log('Ringpop ' + ringpop.whoami() + ' handled ' + key);
                res.end();
            } else {
                console.log('Ringpop ' + ringpop.whoami() +
                    ' forwarded ' + key);
            }
        });

        var port = ports[index] * 2; // HTTP will need its own port
        http.listen(port, function onListen() {
```

```
            console.log('HTTP is listening on ' + port);
        });
    });
}
```

CHAPTER 4

Architecture, Design, and Implementation

4.1 Concepts

4.1.1 Membership Protocol

Ringpop implements a membership protocol that allows nodes to discover one another, disseminate information quickly, and maintain a consistent view across nodes within your application cluster. Ringpop uses a variation of the gossip protocol known as SWIM (Scalable Weakly-consistent Infection-style Process Group Membership Protocol) to disseminate membership updates across the many members of the membership list. Changes within the cluster are detected and disseminated over this protocol to all other nodes.

Ringpop uses the SWIM gossip protocol mechanisms of "ping" and "ping-req". Pings are used for disseminating information and fault detection. Members ping each other in random fashion until they get through the full membership list, rotate the list, then repeat the full round of pinging.

####SWIM Gossip Protocol for Information Dissemination Let's say you have a cluster with two nodes: A and B. A is pinging B and B is pinging A. Then a third node, C, joins the cluster after pinging B. At this point B knows about C, but A does not. The next time B pings A, it will disseminate the knowledge that C is now part of the cluster. This is the information dissemination aspect of the SWIM gossip protocol. ####SWIM Gossip Protocol for Fault Detection Ringpop gossips over TCP for its forwarding mechanism. Nodes within the ring/membership list are gossiping and forwarding requests over the same channels. For fault detection, Ringpop computes membership and ring checksums.

A membership list contains the addresses and statuses (alive, suspect, faulty, etc.) of the instances. It also contains additional metadata like the incarnation number, which is the logical clock. All this information is combined and we compute a checksum from it.

The checksums detect a divergence in the cluster in the event a request is forwarded, or a ping occurs, and the source and destinations checksums differ.

Ringpop retains members that are "down" in its membership list. SWIM manages membership status by removing down members from the list, whereas Ringpop keeps down members in the list allowing the ability to merge a split-brain after a network partition. For example, let's say two clusters form your application. If there isn't a way to identify which nodes were previously faulty or down because the network partition happened during that time, there would be no way to merge them back together.

4.1.2 Consistent Hashing

Ringpop leverages consistent hashing to minimize the number of keys to rebalance when your application cluster is resized. Consistent hashing in Ringpop allows the nodes to rebalance themselves with traffic evenly distributed. Ringpop uses FarmHash as its hashing function because it's fast and provides good distribution. Consistent hashing applies a hash function to not only the identity of your data, but also the nodes within your cluster that are operating

on that data. Ringpop uses a red-black tree to implement its underlying data structure for its ring which provides log n, lookups, inserts, and removals.

Ringpop maintains a consistent hash ring of its members. Once members are discovered to join or leave the cluster, that information is added into the consistent hash ring. Then the addresses of the instances in the ring are hashed.

Ringpop adds a uniform number of replica points per node. To spread the nodes around the ring for a more even distribution, replica points are added for every node within the ring. It also adds a uniform number of replica points so the nodes and the hosts running these nodes are treated as homogeneous.

4.1.3 Forwarding

Ringpop offers proxying as a convenience and can be used to route your application's requests. Traffic through your application is probably directed toward a particular entity in your system like an object with an id. That id belongs somewhere in your cluster on a particular instance, depending on how it hashes. If the key hashes to an instance that did not receive the request, then that request is simply forwarded and everything is taken care of under the hood. This acts like a middleware layer for applications. Before the request even gets to your business logic, it is already routed to the appropriate node.

Ringpop has codified a handle or forward pattern. If a key arriving at instance A hashes to the node, it can process it, otherwise, it forwards it. This information is forwarded using a protocol called TChannel. TChannel is a networking framing protocol developed by Uber, used for general RPC. Ringpop uses TChannel as its proxying channel and transport of choice. It supports out-of-order responses at extremely high performance with benchmarks ranging from 20,000 to 40,000 operations per second.

Ringpop packs forwarded requests as HTTP over TChannel. HTTP is packed into the message that's transported over TChannel when it's forwarded, and then reconstructed on the other side.

Forwarding Requests

As an example, let's say node C joins a ring and now all of the addresses and replica points are evenly distributed around the ring. A, B, and C are pinging one another. The handle or forward pattern peforms a `ringpop.lookup`, gives it the sharding key and gets a destination back. If the destination resolves to A, then A can handle the request; otherwise it forwards it over TChannel transport to its destination.

Note: Eventually, this process will be moved to a Thrift model instead of HTTP.

4.2 How Ringpop Works

4.2.1 Joining a Cluster

1. The first node, A, checks a bootstrap list and finds that no other nodes are running.
2. Next, B starts up and has A to join. B reads the file from disk, then selects a random number of members. It will find A and start to form a consistent hash ring in the background, running within memory in Ringpop.
3. The nodes are positioned along the ring and exchange information with one another, forming a two-node cluster and pinging each other back and forth.

4.2.2 Handle or Forward

Upon arrival of a proxied request at its destination, membership checksums of the sender and receiver will be compared. The request will be refused if checksums differ. Mismatches are expected when nodes are entering or exiting the

cluster due to deploys, added/removed capacity, or failures. The cluster will eventually converge on one membership checksum, therefore refused requests are best handled by retrying them.

Ringpop's request proxy has retries built in and can be tuned using two parameters provided at the time Ringpop is instantiated: `requestProxyMaxRetries` and `requestProxyRetrySchedule` or per-request with: `maxRetries` and `retrySchedule`. The first parameter is an integer representing the number of times a particular request is retried. The second parameter is an array of integer or floating point values representing the delay in-between consecutive retries.

Ringpop has codified the aforementioned routing pattern in the `handleOrProxy` function:

- returns `true` when key hashes to the "current" node and `false` otherwise.
- returns `false` and results in the request being proxied to the correct destination. Its usage looks like this:

```
var opts = {
    maxRetries: 3,
    retrySchedule: [0, 0.5, 2]
};

if (ringpop.handleOrProxy(key, req, res, opts)) {
    // handle request
}
```

4.2.3 Node Statuses

Content coming soon...

4.2.4 Flap Damping

Flap damping is a technique used to identify and evict bad nodes from a cluster. We detect flaps by storing membership update history and penalize nodes when flap is detected. When the penalty exceeds a specified suppress limit, the node is damped. When things go wrong and nodes are removed from the hash ring, you may see a lot of shaky lookups.

As an example, let's say A pings B, and B responds. Then, in the next round of the protocol, A pings B again but this time B is down. Then in the next round, A pings B, but this time B is up again. If there's a bad actor (a slow node that's overwhelmed by traffic), it's going to act erratically. So we want to evict it from the cluster as quickly as possible. The pattern of deviations between alive and suspect/faulty are known as flaps.

We detect flaps by storing the disseminated membership updates as part of the SWIM gossip protocol. When we detect a flap, we penalize the bad actor. Every node stores a penalty for every other node in the cluster. For example, A's view of B is different than C's view of B. When the penalty exceeds a certain suppression limit, that node is damped. That damped status is disseminated throughout the cluster and removed from the ring. It is evicted and penalized so that it cannot join the ring for a specified period of time.

If the damp score goes down and then decays, the problem is fixed and it will not be penalized and evicted from that ring. But if excessive flap exceeds the red line (damping threshold), then a damping sub-protocol is enacted similar to the indirect pinging sub-protocol defined by SWIM.

Say the damp score for B exceeds the red line. A fans out a damp-req request to k random members and asks for their damp score of B. If they also communicate that B is flapping, then B is considered damped due to excessive flapping. A marks B as damped, and disseminates that information using the gossip protocol.

4.2.5 Full Syncing

When a node *a* contacts another node *b*, node *a* sends the checksum of its membership. Node *b* compares the checksums and if they mismatch and there are no more changes to be disseminated, node *b* adds its full membership information in the response. This is called a full sync.

One problem with this technique is that node *a* has not sent its membership information to *b* yet and in some cases this prevents the ring to converge. To make the ring in those cases converge, we need bidirectional full syncs – a way for *b* to get *a*'s full membership as well.

Bidirectional full syncs work as follows. When *b* issues a full sync, *b* also want to know about *a*'s membership. To achieve this *b* sends a join request to *a* and applies the join response to its membership.

In the following example a cluster configuration is disribed that would not converge with the help of only full syncs:

- Partition A in which every node has AB as alive in its membership;
- Partition B in which every node has B as alive in its membership.

When node *a* of partition A contacts node *b* of partition B; *b* finds that the checksums do not match. Therefore a full sync is issued and *b* collects its entire membership for *a*. However, *a* already knows about all the members of *b* and no changes are applied and the ring doesn't converge. In this state, both clusters will have different notions of the membership, and will never converge.

4.2.6 Event Tracing

Ringpop supports live event tracing, which can be sinked to either a TChannel endpoint, or a log-type entity. At present there is only one traceable event: `membership.checksum.update`.

In order to use/test it, you will need to use tcurl and tcat from the tcat branch in tcurl. (Perhaps this will be merged, but for now that's simplest.)

In a window, spawn a ring, via tick cluster, or by hand – whatever makes sense. In our case, we'll assume that a ring of nodes that has a member on localhost:8200.

The listener (tcat) looks like this:

```
    /path/to/tcurl/tcat.js -p localhost:4444 ringpop foobar
```

Where `foobar` is an arbitrary endpoint name (like a URL path). `tcat` will listen on localhost port 4444, service `ringpop`, and accept and log all messages to the `foobar` endpoint.

In another window, register the event tracer at a single member, in this case localhost:8200:

```
tcurl -p 127.0.0.1:8200 ringpop /trace/add -3 '{"event": "membership.checksum.update", "expiresIn": 1
```

The command should return a result of `OK`. (The ringpop member @ 8200 may emit a socket closed error – this is due to an assumption in tcurl, but is not a problem.)

Now you should have all membership checksum updates log to the tcat window. To test it, kill a ringpop member NOT at port 8200. That should generate a message. Restart it. That should also generate a message.

My output looked like:

```
[benfleis] ~/src/ringpop/node $ ~/src/tcurl/tcat.js -p localhost:4444 ringpop foobar
{"id":"127.0.0.1:8200","address":"127.0.0.1:8200","timestamp":1444129621728,"checksum":107935330,"mem
{"id":"127.0.0.1:8200","address":"127.0.0.1:8200","timestamp":1444129623663,"checksum":1332854908,"me
```

4.2.7 TChannel

Content coming soon...

TChannel is a network multiplexing and framing protocol for RPC. Some of the characteristics of TChannel:

- Easy to implement in multiple languages, especially JavaScript and Python.
- High performance forwarding path. Intermediaries can make a forwarding decision quickly.
- Request/response model with out-of-order responses. Slow request will not block subsequent faster requests at head of line.
- Large requests/responses may/must be broken into fragments to be sent progressively.
- Optional checksums.
- Can be used to transport multiple protocols between endpoints, e.g., HTTP + JSON and Thrift

Components

- tchannel-node: TChannel peer library for Node.js.
- tchannel-python: TChannel peer library for Python.
- tchannel-golang: TChannel peer library for Go.
- tcap: TChannel packet capture tool, for eavesdropping and inspecting TChannel traffic.
- bufrw: Node.js buffer structured reading and writing library, used for TChannel and Thrift.
- thriftrw: Node.js Thrift buffer reader and writer.
- thriftify: Node.js Thrift object serializer and deserializer with run-time Thrift IDL compiler.

4.3 Extensions

Ringpop is highly extensible and makes possible for a multitude of extensions and tooling built around it. Here are the libraries that extend Ringpop.

4.3.1 Sharding

Content coming soon...

4.3.2 Actor Model

Every actor in the system has a home (a node in the cluster). That node receives concurrent requests for every actor. For every actor, there is a mailbox. Requests get pulled off the mailbox one by one. Processing a request may result in new requests being sent or new actors being created. Each request that's processed one by one may result in some other request to another service, or a request for more actors to be spun up.

4.3.3 Replication

Building Redundancy with Ringpop.

Reliable Background Operations

Content coming soon...

Leader Election

Content coming soon...

4.3.4 Partition Recovery

As of ringpop-go DEV and ringpop-node DEV, ringpop is able to detect and heal partitions. In a nutshell, it works as follows:

1. Query the Discover Provider for a list of hosts that should be in the cluster.
2. Unseen and `faulty` hosts are pinged, and, if successful, joined to the cluster.

For detailed design on how that works, see Partition Healing page.

CHAPTER 5

Partitions

In the original implementation of ringpop, if a cluster is split to multiple partitions, nodes in each partition declare each other as faulty, and afterward will no longer communicate. Ringpop implemented support for merging the partitions, which we call `healing`.

5.1 Partition Healing – basic algorithm

In order for two partitions to heal, the algorithm does the following, periodically (some details are omitted for brevity; exact algorithm can be found in the code comments in the implementation):

1. Randomly select a `faulty` node.
2. Send it a `/join` request, get its membership list.
3. If the local and retrieved lists are incompatible (merging them will introduce new faulties), mark all incompatible nodes suspect. When receiving this change the respective node will reassert that it is actually alive and update its incarnation number making it compatible for merge.
4. If the local and retrieved lists are compatible (merging them will not introduce new faulties), merge the membership list with the local node's membership and disseminate the changes.

We test this feature in 3 ways:

1. Unit tests asserting the correct behavior.
2. Integration tests, which will be the same for Go and Node implementations, checking the behavior of a node in isolation.
3. Manual acceptance test to see partitions actually getting healed.

Further down, we will talk about how to manually create a partition and confirm it heals itself.

5.2 Forming a partition

With the current implementation of tick-cluster, it is non-trivial to form a partition. To understand why, we need to understand how connections are established.

5.2.1 Port Allocation

A ringpop instance opens a local tchannel socket (=listening tcp socket) to accept incoming connections from other ringpops. By default, on a 2-node tick-cluster, this is `127.0.0.1:3000`. Let's call it instance `a`. For instance `a` to es-

tablish a connection to instance b (127.0.0.1:3001), instance a will open an ephemeral port, e.g. 43323, to connect to instance b. This connection, from 127.0.0.1:43323 (a) to 127.0.0.1:3001 (b) is used for messages initiated by node a. The other connection (example below), from 127.0.0.1:36113 (b) to 127.0.0.1:3000 (a), is used for messages initiated by b. Here is a snapshot of lsof from a two-node cluster:

```
root:/# lsof -Pnni
COMMAND PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
node     64 root   10u  IPv4 217924      0t0  TCP 127.0.0.1:3000 (LISTEN)
node     64 root   11u  IPv4 217925      0t0  TCP 127.0.0.1:43323->127.0.0.1:3001 (ESTABLISHED)
node     64 root   12u  IPv4 217926      0t0  TCP 127.0.0.1:3000->127.0.0.1:36113 (ESTABLISHED)
node     66 root   10u  IPv4 219916      0t0  TCP 127.0.0.1:3001 (LISTEN)
node     66 root   11u  IPv4 219917      0t0  TCP 127.0.0.1:36113->127.0.0.1:3000 (ESTABLISHED)
node     66 root   12u  IPv4 219918      0t0  TCP 127.0.0.1:3001->127.0.0.1:43323 (ESTABLISHED)
root:/#
```

Armed with this knowledge, we can try to make a partition.

5.2.2 Manually forming a partition

The naïve approach to make a partition between a and b is to block incoming connections from and to port 3000: then no packet will leave a, and we will have a partition. However, this misses the fact that ephemeral connections are used for relaying traffic between nodes, and, in this case, connection from 127.0.0.1:43323 (a) to 127.0.0.1:3001 is established and... misses the firewall! We could block port 3001 too, but, with more nodes, that would create a cluster with N partitions (N being the number of nodes) – not what we want. In our example, we want two partitions.

With that in mind, a bit more sophistication in firewall rules is required. To easily create a partition in tick-cluster locally, we created tools/make_partitions, which, by reading the state of the connections from lsof, will emit iptables/pf commands accordingly.

In the example above, firewall rules to create a partition will look as follows (OS X):

```
$ sudo lsof -Pnni | ./tools/make_partition 3000 3001 --platform darwin
block drop in proto tcp from 127.0.0.1 port 3000 flags S/S
block drop in proto tcp from 127.0.0.1 port 3001 flags S/S
block drop in proto tcp from 127.0.0.1 port 43323 to 127.0.0.1 port 3001
block drop in proto tcp from 127.0.0.1 port 3001 to 127.0.0.1 port 43323
block drop in proto tcp from 127.0.0.1 port 36113 to 127.0.0.1 port 3000
block drop in proto tcp from 127.0.0.1 port 3000 to 127.0.0.1 port 36113
```

Linux:

```
$ sudo lsof -Pnni | ./tools/make_partition 3000 3001 --platform linux
*filter
-A INPUT -p tcp -s 127.0.0.1 -d 127.0.0.1 --tcp-flags RST RST -j ACCEPT
-A INPUT -p tcp --syn -m state --state NEW -d 127.0.0.1 --dport 3000 -j REJECT --reject-with tcp-rese
-A INPUT -p tcp --syn -m state --state NEW -d 127.0.0.1 --dport 3001 -j REJECT --reject-with tcp-rese
-A INPUT -p tcp -s 127.0.0.1 --sport 43323 -d 127.0.0.1 --dport 3001 -j REJECT --reject-with tcp-rese
-A INPUT -p tcp -s 127.0.0.1 --sport 3001 -d 127.0.0.1 --dport 43323 -j REJECT --reject-with tcp-rese
-A INPUT -p tcp -s 127.0.0.1 --sport 36113 -d 127.0.0.1 --dport 3000 -j REJECT --reject-with tcp-rese
-A INPUT -p tcp -s 127.0.0.1 --sport 3000 -d 127.0.0.1 --dport 36113 -j REJECT --reject-with tcp-rese
COMMIT
```

To sum up:

- New connections to the listening ports (3000, 3001) will be blocked. This prevents tchannel to re-open new valid connections.
- Relevant existing connections will be terminated (e.g. 3000 to ephemeral ports).

- Linux only: for the above to work, the firewall needs to explicitly accept `RST` packets.

During the partition, new connections to the nodes will be impossible to make. This is important to keep in mind when using `ringpop-admin`: **invoke ringpop-admin before forming the partition**.

Armed with background how this works, we can go and make a local partition:

5.2.3 Start the tick-cluster

In this example, we use Node version of ringpop, but we can use `testpop` from go too:

```
$ ./scripts/tick-cluster.js -n 4 ./main.js  # node
```

5.2.4 Open the ringpop-admin to observe the cluster state

We shall open `ringpop-admin partitions` and `ringpop-admin top` before making the partition. This way, the "management" connections will be open and status will be visible during the partition:

```
$ ringpop-admin top 127.0.0.1:3000
$ ringpop-admin partitions -w 1 127.0.0.1:3000  # other terminal
```

`ringpop-admin top` will show something like this:

```
Address          P1
127.0.0.1:3000   alive
127.0.0.1:3001   alive
127.0.0.1:3002   alive
127.0.0.1:3003   alive
1 of 4
```

`ringpop-admin partitions` will show a single partition, updated every second:

```
10:27:09.615   Checksum    # Nodes   # Alive   # Suspect   # Faulty   Sample Host
               192859590   4         4         0           0          127.0.0.1:3000
10:27:10.607   Checksum    # Nodes   # Alive   # Suspect   # Faulty   Sample Host
               192859590   4         4         0           0          127.0.0.1:3000
```

5.2.5 Start the partition

First, check how the firewall rules would look like before applying them to the firewall (optionally, you can pass `--platform=darwin` or `--platform=linux` to the `make_partition` script:

```
$ sudo lsof -Pnni | ./tools/make_partition 3000,3001 3002,3003
```

If you are happy with the output, apply the rules:

OS X:

```
$ sudo lsof -Pnni | ./tools/make_partition 3000,3001 3002,3003 | sudo pfctl -emf -
```

Linux:

```
$ sudo lsof -Pnni | ./tools/make_partition 3000,3001 3002,3003 | sudo iptables-restore
```

In a few seconds, you should see output from `tick-cluster` that some of the nodes aren't able to ping each other. Let's verify we actuall have a partition.

5.2.6 Checking in the tools

On partition, `ringpop-admin top` (opened before the partition) should display something like this:

```
Address          P1       P2
127.0.0.1:3000   faulty   alive
127.0.0.1:3001   faulty   alive
127.0.0.1:3002   alive    faulty
127.0.0.1:3003   alive    faulty
```

`ringpop-admin partitions` (opened before forming a partition) shows a more high-level view:

```
10:37:04.878   Checksum     # Nodes   # Alive   # Suspect   # Faulty   Sample Host
               400620880    2         2         0           2          127.0.0.1:3002
               3283514511   2         2         0           2          127.0.0.1:3000
```

That's it, we have a partition! To break it, we need to wipe the firewall rules:

- OS X: `pfctl -f /etc/pf.conf`.
- Linux: `iptables -F`.

... and wait for partition healing to kick in.

5.2.7 Final remarks

- `tools/make_partition` can only create two partitions. It can work with arbitrary partition sizes; for usage, run `tools/make_partition --help`.
- `tools/make_partition` is not intended to be used in an automated way. See `--help` to learn about the limitations.

CHAPTER 6

References

Learn more about key concepts related to Ringpop.

6.1 FAQ

6.2 Glossary

6.2.1 A

- **Actor model**: Concurrent computation model used by Ringpop that allows messages to arrive concurrently, then processed one by one. Messages are placed in a mailbox based on the sharding key, and then processed one by one. Each request that's processed one by one may result in some other request to another service, or a request for more actors to be spun up. Alive: A membership status signifying the node is healthy, and not suspect, faulty, or damped.

6.2.2 B

- **Bad actor**: A slow node that's overwhelmed by traffic.

6.2.3 C

6.2.4 D

- **Damped**: Flap damping is a technique used to identify and evict bad nodes from a cluster. Flaps are detected by storing membership update history and penalize nodes when flap is detected. When the penalty exceeds a specified suppress limit, the node is damped. The damped status is disseminated throughout the cluster and removed from the ring.

6.2.5 E

6.2.6 F

- **Flap damping**: Flap damping is a technique used to identify and evict bad nodes from a cluster.
- **FarmHash**: Hashing function used by Ringpop.

- **Faulty**: A state of the node that is reached after a defined "suspect" period, where a node is unstable or not responding to pings from other nodes. A suspect period will begin, and if it ends with the node not recovering, the node is considered faulty and is removed from the ring.

6.2.7 G

- **Gossip**: A type of protocol where nodes disseminate information about each other using pings.

6.2.8 H

- **Handle or forward**: This is Ringpop's forwarding approach. If a key hashes to an instance that is not the one that received the request, then that request is simply forwarded to the proper instance and everything is taken care of under the hood. This acts like a middleware layer for applications that before the request even gets to your business logic, it is already routed to the appropriate node.
- **Hash ring**: Ringpop leverages consistent hashing to minimize the number of keys to rebalance when your application cluster is resized. Ringpop's consistent hashing allows the nodes to rebalance themselves and evenly distribute traffic. Ringpop maintains a consistent hash ring of its members. Once members are discovered to join or leave the cluster, that information is added into the consistent hash ring. Then the instances' addresses along that ring are hashed.

6.2.9 I

6.2.10 J

6.2.11 K

6.2.12 L

6.2.13 M

- **Membership list**: Ringpop uses a variation of SWIM to disseminate membership updates across the members of a membership list, which contains additional metadata like the incarnation number, instances' addresses, and status (alive, suspect, faulty, etc.). Members ping each other in random fashion until they get through the full membership list, rotate the list, then repeat the full round of pinging.
- **Multi-cast**:

6.2.14 N

6.2.15 O

6.2.16 P

- **Ping**: Ringpop uses pings to disseminate information and for fault detection. Members ping each other in random fashion until they get through the full membership list, rotate the list, then repeat the full round of pinging.

6.2.17 Q

6.2.18 R

- **Replica points**: Ringpop adds a uniform number of replica points per node to spread the nodes around the ring for a more even distribution. Ringpop also adds a uniform number of replica points so the nodes and the hosts running these nodes are treated as homogeneous.
- **Ringpop**: Ringpop is a library that brings application-layer sharding to your services, partitioning data in a way that's reliable, scalable and fault tolerant.
- **Ringpop forwarding**:

6.2.19 S

- **SERF**: Gossip-based membership that exchanges messages to quickly and efficiently communicate with nodes.
- **Sharding**: A way of partitioning data, which Ringpop does at the application layer of your services in a way that's reliable, scalable and fault tolerant.
- **Suspect**: A state of the node where it is unstable or not responding to pings from other nodes. If nodes stay suspect during the pre-defined suspect period without recovering, it will then be considered faulty and removed from the ring.
- **SWIM**: Scalable Weakly-consistent Infection-style Process Group Membership Protocol

6.2.20 T

- **TChannel**: TChannel is a network multiplexing and framing protocol for RPC. TChannel is the transport of choice for Ringpop's proxying channel.

6.2.21 V

6.2.22 W

6.2.23 X

6.2.24 Y

6.2.25 Z

6.3 Use Cases

6.4 Papers

- BGP Route Flap Damping
- Dynamo: Amazon's Highly Available Key-value Store
- Efficient Reconciliation and Flow Control for Anti-Entropy Protocols
- Epidemic Broadcast Trees

- FarmHash
- Riak
- SWIM Presentation Slides by Armon Dadgar from Hashicorp
- SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol
- TChannel
- The Accrual Failure Detector
- Time, Clocks, and the Ordering of Events in a Distributed System

6.5 Presentations

CHAPTER 7

Community

7.1 Google Group

7.2 Contributing

7.3 License

Ringpop is available under the MIT license. See the LICENSE file for more info.