Building a Fault-Tolerant Distributed System with zookeepertcl

Garrett McGrath

Introduction

`zookeepertcl` provides a natural Tcl interface to the Apache Zookeeper API. Although incredibly versatile and powerful, the API has a surface-level simplicity that potentially suggests otherwise. To allay this potential misunderstanding, the Zookeeper project has documented a number of recipes utilizing the API for accomplishing some of the more common coordination tasks in distributed systems. These recipes, while decidedly valuable and important for highlighting the capabilities of Zookeeper, necessarily focus only on using its API profitably. The API and the recipes leave out the many crucial design decisions encountered by a developer trying to utilize one of the recipes as an essential component in a production system. With this in mind, the discussion that follows aims to to bridge the gap between a generic recipe and an actual implementation by detailing a concrete instance of a Tcl-driven, Zookeeper-based system.

Scope of the System

In order to contain the massive scope of this topic, this paper will consider only a limited, but still useful system design for illustrating the implementation specifics left out of the general Zookeeper recipes. This system consists of an arbitrary number of nodes, or machines, and a number of components, or processes, that must all execute simultaneously for the system to function. It can be assumed that the number of components is always equal to or greater than the number of nodes available. As a further restriction, each component is only allowed to run on a single node at any given time. Not all faults can be handled, although the system is fault tolerant to several common failures. In particular, the system cannot handle so-called "Byzantine" failures where nodes produce arbitrary failures. Instead, network failures and node crashes can be dealt with. Primarily, this system can handle some but not all nodes crashing and still remain available, although perhaps with degraded service and with no consistency guarantees made since the internal operation of the components is not specified.

On each of the nodes a daemon will run which communicates with Zookeeper using the `zookeepertcl` library and supervises the execution the system's components. It is of course possible to build in Zookeeper connectivity and application logic to each of the components

individually, but for reasons to be discussed later on, particularly regarding intentional restarts, the daemon-wrapper approach is preferred. This Zookeeper-based daemon will use the leader election recipe in the Zookeeper documentation to determine which components it will run and supervise. In addition, the leader election recipe also furnishes it with a clear procedure for taking over leadership in the event that another node crashes. To summarize, the Zookeeper-based daemon runs on all the nodes in the system and is responsible for, through the leader election recipe, determining which components to run and providing fault tolerance when the leader on another node crashes. In other contexts, e.g., in Kafka, Raft or Zookeeper itself, leader election is used so that a single task requiring all the nodes in the system working together, such as maintaining an append-only log, can grant one node a special role during that procedure. For the system described herein, however, the leader of a component designates a particular node responsible for running it and supervising it; it does not mean that the other nodes are cooperating with it to help run the component, only that the other nodes can take over running and supervising it if the current leader should fail. Lastly, configuration data for each component is stored in Zookeeper. This data specifies how the daemon should run the component, i.e., it contains the command line arguments needed to launch and then supervise it.

Zookeeper Basics

Zookeeper, a software project originally developed at Yahoo but currently managed by Apache is a server written in Java. It is meant to run as a high-availability service across a cluster of machines and makes many coordination tasks in distributed systems simple to implement. Although there are many differences among distributed systems, there are many shared tasks that they tend to perform. For instance, queuing tasks for nodes to work on in parallel, locking a resource so that only one process can write to it at any given time, agreeing on the value of a shared variable, or electing one node as the leader of a task. Cleverly, rather than providing the ability to carry out any of these tasks directly with pre-baked implementations, Zookeeper instead exposes a very simple, file-system like API that makes these tasks possible with a minimum of effort and with a tested and reliable implementation.

Once a connection to Zookeeper has been made, so-called znodes can be created. znodes are analogous to a file: they have a unique path, relative to the root path `/`, and contain both data and metadata. The data portion of a znode can contain anything but is not meant to be large (typically 1MB or less); the meta-data includes a version number (each modification to the znode increments the version), a number of timestamps, along with an access control list (ACL) for security purposes. Additionally, znodes come in two flavors: durable or ephemeral. Durable znodes last until they are explicitly deleted whereas an ephemeral znode only lasts for the duration of the connection that created it. Moreover, it is also possible to create a sequential

znode whose path name, at creation time, is appended with a monotonically increasing sequence number, e.g., a request to create a sequential znode under `/example_` would result in a znode whose full path is something like `/example_00001`.

Zookeeper's central data central data structure, the znode, can be interacted with through a small number of provided API operations. A znode can be created, its data and metadata can be queried, its data can be changed, its existence can be questioned, the name of its child znodes can be requested, and it can be deleted. Each of the aforementioned operations can also be associated with a watch callback. Watch callbacks fire once and only once whenever the operation they are defined on occurs. For instance, if a child watch is placed on a znode and a child znode is created the watch will fire. It is important to keep in mind that once a watch fires, it is no longer there and must be reset every time it needs to be in place. Also, watch callbacks are like ephemeral nodes: they only remain in effect for the duration of the connection that set them. All discussion below, though, assumes that any watch callbacks are properly reset when necessary.

zookeepertcl Peculiarities

Using the Zookeeper API in Tcl is straightforward. The `zookeepertcl` library provides a fast, efficient, easy-to-use and well documented wrapper over the Zookeeper C client included in the official Apache source code. Each of its commands supports both a synchronous and an asynchronous method of operation. Notably, though, the synchronous operations still use the Tcl event loop to pass data between the Zookeeper C library and zookeepertcl, so it will not block events from being processed. Given, then, that the synchronous and asynchronous operations are functionally equivalent, either can be used for any situation depending on circumstance and taste.

Despite its ease of use, there are some peculiarities around connection establishment that should be kept in mind when using the library. When connecting to Zookeeper, one of the required arguments is a timeout in milliseconds for establishing a connection. It might be assumed that if an initial connection is not established before the passed in timeout value, then `zookeepertcl` will throw an error or otherwise indicate that no connection could be made. That is decidedly not the case (unless the hostname cannot be resolved in which case an error is thrown). Behind the scenes, the C client will continually attempt to reconnect to Zookeeper after no connection has been made within the allotted timeout period. This connection re-attempt loop is entered whenever a connection cannot be established whether or not an initial connection was ever made. Therefore, additional code needs to be written to check for the failure to obtain an initial connection to Zookeeper. Even in synchronous mode, without this check for connection

failure, `zookeepertcl` will try forever to obtain a connection. An example of the described connection failure check is found below.

```
proc connect_to_zk {host timeout} {
    zookeeper::zookeeper debug_level none
    try {
        zookeeper::zookeeper init zk $host $timeout \
          -async connect_callback
        after $timeout {set ::zkConnected 0}
        vwait ::zkConnected
        return $::zkConnected
    } on error {args} {
        return 0
    }
}

proc connect_callback {args} {
    set ::zkConnected [expr {[zk state] eq "connected"}]
}

set zkHostString "broken.host:2181"
set zkTimeout 3000

if {[connect_to_zk $zkHostString $zkTimeout]} {
    # connection established
    puts "Connected!"
} else {
    # no connection obtained
    puts "No connection"
}
```

Leader Election in zookeepertcl

The fault-tolerant distributed system discussed herein relies on the leader election recipe provided in the official Zookeeper documentation. Rather than merely repeat it verbatim, a bare-bones implementation in Tcl is provided below with explanatory comments throughout. In this case, code speaks louder than words.

```
# assume a successfully connected Zookeeper object named zk
```

```
# step 1: Create znode z with path "ELECTION/n_" with both SEQUENCE
and EPHEMERAL flags;
# assume that the $electionRoot already exists
set electionRoot [file join / ELECTION]
set electionZnodePath [file join $electionRoot "n_"]
set z [zk create $electionZnodePath -ephemeral -sequence]

# step 2: Let C be the children of "ELECTION", and i be the sequence
number of z;
# zk children returns znode paths relative to that passed in as its
znode parameter
set c [zk children $electionRoot]

# step 3: Find the leader
proc election_cmp {lhs rhs} {
    set lhsNum [scan [lindex [split $lhs _] end] %d]
    set rhsNum [scan [lindex [split $rhs _] end] %d]

    if {$lhsNum < $rhsNum} {
        return -1
    } elseif {$lhsNum == $rhsNum} {
        return 0
    } else {
        return 1
    }
}

set sortedVotes [lsort -command election_cmp $votes]
set leader [lindex $sortedVotes 0]

# step 4: Watch for changes on "ELECTION/n_j", where j is the largest
sequence number such that j < i and n_j is a znode in C;

proc watch_next_node {sortedVotes} {
    # only matters if z is not the leader
    # if z is the leader, do not need to
    # watch anything to comply with the recipe
    set zVotePosition [lsearch $sortedVotes $z]
    set nodeToWatch [lindex $sortedVotes [expr {$zVotePosition - 1}]]
    if {$nodeToWatch in $sortedVotes} {
```

```
        set n_j [file join $ELECTION $nodeToWatch]
        zk exists n_j -watch election_change
    }
}

proc election_change {eDict} {
    # figure out who should be the next leader
    # if z is the new leader, take over
    # otherwise watch the next n_j in the
    # remaining znodes in C
}
```

Implementation Decisions

With the preliminary explanations out of the way, the sections that follow discuss some implementation details left out of the leader election recipe. These details are encountered by anyone attempting to build a version of the fault-tolerant distributed system discussed in this paper.

Abdication

With the basic system design in place, one of the first issues that comes up is that of timing. Given that leader election for each component process will run on a multitude of hosts, it is highly probable that a particular component's election will not occur at the same time on each node. Instead, it is entirely possible and much more likely that the first node to run leader election will win the election for every component. Depending on the size and resource consumption of each component, this might not be a problem, but it is certainly less than ideal to have a single node running every constituent component in a distributed system. Instead, there needs to be some way of preventing this situation from occurring, and to ensure that each node capable of doing so can participate in the running of processes.

Undoubtedly a number of strategies might be employed to combat a single node from taking all of the work, although not every possibility will be explored here. Instead, a single strategy, named abdication, is suggested. Under abdication, when a node wins leadership of a component, a decision must be made as to whether this is actually desirable. In addition, every leader sets a watch callback on the children of its election znode and runs through the same decision procedure whenever it fires. The decision on whether to retain leadership can take an arbitrary amount of information into account, but, for the sake of illustration, the simplest would be whether or not continuing as the leader retains a fair distribution of work across all the nodes

in the distributed system.[1] If accepting leadership still retains fair distribution, then no abdication is necessary. On the other hand, if accepting leadership would assign an unfair distribution of work, then leadership is abdicated.

Abdicating leadership is a straightforward procedure but can have lots of side effects for other portions of the system, so it is necessary to detail it explicitly. During abdication the node giving up leadership deletes its ephemeral, sequential znode created for voting (it was named `z` in the previous section's code). Doing this deletion will trigger a watch callback for one of the other nodes and promote it to leader. After deletion, the newly abdicated node should once again put itself back in the running for this component by redoing election. However, this time, since another node has already taken leadership of the process, this will not reintroduce unfair distribution of work. Moreover, redoing election is important so that no component in the system is left without a leader when there are nodes available for taking on additional work.

Intentional Restarts and Stops

After addressing the potential for a minority of nodes to take all the available work and crowd out the ability of other machines in the system to contribute their compute resources, another salient issue that arises is how to handle intentional restarts and stops of components without giving up leadership. Without any modification to the leadership recipe, when restarts and stops occur this will signal to the rest of the system that the leader has died and needs to be replaced. In the case of unintentional restarts and stops, this is exactly what is desired, but when an admin wants a restart or stop to occur it is not always desirable to redo election. For instance, throughout the course of a distributed system's execution, it is quite likely that there will be a need for individual components, either in isolation or en masse, to have their code upgraded. When this happens (assuming a hot swap cannot occur), the overhead of leader election is often unwanted. Turning to the justification for intentional stops, it is possible that a particular component is malfunctioning, no matter what node it runs on. In that case, debugging is required, and, until a fix is discovered, no node should run the given component. Under this scenario when the component is stopped, no other node should attempt to run it, so it needs to be stopped without triggering another election.

Adding the ability to intentionally restart or stop a component without giving up leadership can certainly be done in a number of ways. Considering, though, the previously discussed strategy of abdication, a simple method suggests itself. Under abdication, every leader has a watch callback set on its child znodes that no other node will have. Therefore, a simple

---

[1] Again, like with the amount of information used to decide on abdication, what constitutes fair distribution is open to a multitude of definitions. For the sake of simplicity in the discussion that follows, assume that for the current node, i.e., one that has won leadership of a component and is considering abdication, fair distribution is calculated as `number of components / number of nodes`.

method of communicating something exclusively to the leader is to create a child znode under the root znode for component's election. Either the path or the data of this child znode can contain instructions for the leader to restart or stop the component. In the case of a restart, when the component is back up it can delete the child znode that told it to restart; in the case of a stop, another mechanism, perhaps a signal handler, will be needed to get the component to resume execution, but once that occurs, the child znode, likewise, can be deleted. Given that a wrapper program has been recommended for each component, this method offers an easy way of intentionally starting or stopping any component in the system. For added protection, particular ACL schemes can be enforced for the child znodes used to restart or stop a component.

Config Changes

Provided the ability to restart a component without giving up leadership, it is then relatively simple to watch for config changes and selectively restart any affected components. This allows for seamlessly changing the config values in a running system without anything more than altering the data in a znode. Despite this genuine simplicity, a bit of caution is in order. While the simplest implementation would have the watch callback immediately restart any component whose config changed, this eagerness is not advisable. For one thing, it is often common for multiple config znodes to be changed at once. With that being the case, a situation to avoid is where one config znode changes, a watch callback fires, a component is restarted as a result and, then, while it is restarting, another config znode changes and this process repeats. To prevent this, a delay should be introduced before a restart occurs, with the delay extended every time another change is detected.

Zookeeper Connection Loss

When a connection to Zookeeper has been lost for a node leading one or more components, it is fairly straightforward to figure out what needs to be done: all components currently being led need to be stopped immediately and the next node in line for leadership must takeover.[2] What it means for a connection to Zookeeper to be lost is not straightforward, however. At any given time, a Zookeeper connection can be in one of several states: `closed`, `connecting`, `associating`, `connected`, `expired`, `auth_failed` and `unknown`. `connected` unambiguously means that a usable connection has been established; conversely, `closed` and `expired` unambiguously mean that the connection has been lost. As for the

[2] Nothing in distributed systems is easy and in this situation, extra care needs to be taken to ensure that the new leader does not start running the component before the old leader shuts it down. Detailing methods for doing this is outside the scope of this discussion, but, in general, this needs to be built into the component itself rather than part of Zookeeper API operations.

others, `auth_failed` relates to ACLs and does not affect the network connection although it will preclude performing any unallowed actions on nodes requiring an ACL with an auth scheme; `unknown` should be considered a broken connection and treated like `closed` and `expired`; lastly, `associating` and `connecting` represent an in-between state between `connected` and `closed` and require the most complicated handling.

As mentioned in a previous section on peculiarities with `zookeepertcl` connection establishment, the official C client will always attempt to re-establish a connection as long as the hostnames in its connection string successfully resolve. Connections in the `associating` and `connecting` state, then, will remain there until the C client can successfully transition to the connected state. Typically a connection will transition to `associating` or `connecting` in a clustered Zookeeper setup when one of the servers cannot be reached. When this is a short lived event, this is quite desirable and can help avoid the overhead of another election. Otherwise, if the network failure lasts for any substantial amount of time, having a component's leader go into an endless reconnect loop in a distributed system where all components need to be running with minimal down-time is unacceptable. Instead, connections in those states need to be watched so that they do not loop indefinitely. To fix this, a Tcl `after` callback can be scheduled to allow `zookeepertcl` some small amount of time to become connected. If the `after` callback detects that no connection has been made within the time allowed, then the component should be shut down and another node allowed to become the leader.

Detecting Restart Loops

Related to the prevention of a connection retry loop is the issue of detecting restart loops. While a distributed system with automatic failover is quite powerful, its strength, i.e., the ability to automatically respond to the loss of leadership, can become a crushing weakness without the proper precautions. Imagine the situation where a node with at least one component temporarily loses its connection to Zookeeper without obtaining a connection in sufficient time to retain leadership. When that happens, another node will take over leadership. However, it could happen that the new leader suffers the same fate as the old one, and that the same fate happens to the next leader after that ad nauseam. Should this happen, any components on the failing leader will get passed around from one node to the next without ever being able to make progress.

This situation could also occur in a number of other situations. For instance, it could happen to a component that, for whatever reasons, cannot run successfully on its current leader's machine. If the system chooses to give up leadership when a component stops unintentionally, then another node will take over leadership. But, assuming every other node is already at full capacity, then every other node other than the one where the component does not run properly would abdicate leadership once the broken node re-elects itself. Once again a situation arises

where a component would get stuck in a restart loop. To prevent this situation, the system must track when and how many times it attempts to assume leadership for a given component. If a threshold number of attempts happen within a threshold amount of time, then a policy can enforce the proper behavior for the system based on the component itself and its role in the system overall.