CPSC 551 - Project 3

Justin Chin, Daniel Miranda

December 16, 2019

1 Introduction

Throughout the semester we have been building a distributed microblogging platform using tuplespaces. The underlying model has a number of users each with an individual tuplespace that they interact directly with. Any changes that they make to their local tuplespace (i.e write and take operations), should be propagated and replicated to all other clients in the microblog group. That is, there should be a shared global view of the tuplespace contents.

The second project saw us implementing this model in two different ways. In the first approach, we used a centralized architecture where each user in the microblog platform had an individual tuplespace/adapter pair. This approach had the benefit of relatively easy discovery. As new tuplespaces join the multicast group, the nameserver records their endpoints in its own tuplespace. A command line client then connects to this naming server at a well-known address, retrieves a list of client-address bindings, and performs operations directly on the users tuplespaces. Assuming no failures, this is obviously a correct approach, such that all the operations (if they succeed), will occur in the same order on each tuplespace. The command line client we wrote for interacting with the microblog platform essentially performs the role of a centralized coordinator for every event, as it connects directly to each user's tuplespace and performs the operation. The system has a single view, that of the client's. This becomes a little more muddied when we introduce multiple clients as we have introduced race conditions, because there was originally no lock mechanism to enforce mutual exclusion.

The second approach had us implementing a decentralized architecture, by removing the naming and log/recovery server. Instead of allowing those servers to be individual points of failure, we merged the two facilities into a middleware layer (i.e the Tuplespace Manager). Users would interact directly with their own Tuplespace Manager, which would apply client operations to the tuplespace. The tuplespace manager would also listen for multicast traffic from the events announced by the tuplespaces in the multicast group, intercept, and replicate them locally.

Through the course of both implementations, numerous problems arose resulting from flaws in the design. The first approach struggles from having multiple points of failure (i.e the nameserver), where a single link failing brings the entire system into an unusable state. The second approach suffers from a separate, yet ultimately fatal flaw, wherein a single message sent into the platform will bounce around indefinitely between all the tuplespaces as they multicast their events, and respond in turn to those events, ad infinitum.

2 Flaws in the previous designs

2.1 Duplication of events

In both approaches of Project II we encountered duplication of events/actions that materialized from our implementations of the Recovery Server (i.e. log file) for the first approach, and Replication in the second approach.

2.1.1 Recovery Server (log file) problem:

The recovery server listened for 'write' and 'take' notifications from the multicast group and logged each one to a log file, in essence, taking a record/history of all events transpiring within the multicast group. The problem arises when a tuplespace is started/restarted and in need of being brought up to date. Upon recovery of a newly started tuplespace the recovery server would playback all the events in its log to that tuplespace bringing it up to date and consistent with the rest of the group. Initially, this would work as expected, but only once. From then onwards, the log would be corrupted because the recovery server logs events indiscriminately. Events that were being played back for the newly joining tuplespace would be performed at that tuplespace, and by design the tuplespace would multicast that action to the recovery server where it was logged as if that tuplespace were performing them as a normal operation rather than a recovery operation. Although the first tuplespace having a recovery performed upon it would be consistent with the rest of the group, any subsequent recovery of tuplespaces would yield duplicate entries in its tuplespace.

2.1.2 Replication problem:

In the replication approach the problem was more severe when it came to duplicating entries. In this approach we used tuplespace managers at each tuplespace that would listen for multicast traffic for 'write' and 'take' events. Upon of the receipts of these notification the action would be taken locally (active replication) which in theory would keep all the tuplespaces consistent. However, as mentioned previously, there was no mechanism to distinguish entries that had already been seen, no way to uniquely identify them. In turn, as soon as a 'write' event had been performed at one tuplespace, a multicast would be sent and received by the other tuplespace managers in the group which would perform the operation locally. This now led to each of the other tuplespaces multicasting their write event, which in turn would cause them to again multicast the event after performing it, so on and so forth. This led to perpetual writing in the system and rendering the entire system useless as each of the processes resources were being used to continuously write the same operation over and over. In a similar fashion if a 'take' event was performed, it would cause the same issue, however in this case the system would hang due to the fact that the operation would quickly exhaust the tuplespaces of entries matching the 'take' criteria and all processes would eventually wait in a blocking state for a tuple to appear that would match the criteria of the 'take' operation. This too would render the system useless.

2.1.3 What triggers the problem, and how to reproduce it:

1. Recovery Server (log file): The issue was triggered by performing more than one recovery. The first recovery, and only the first one, would leave the tuplespaces in a consistent state because the log file at that time would still be in an untainted state. However, after the first recovery, the log file would be corrupted and never again be correct.

2. Replication: The issue of perpetual writes/takes of tuplespaces could be triggered from the onset of the very first operation performed on it. A single event would immediately put it into a state of constant operation (write) or constant waiting (take).

2.1.4 What impact the problem has on the system:

1. Recovery Server (log file): The system would continue to operate. However, the tuplespaces would not be in a consistent state, nor would they ever reach consistency amongst them (except for the first recovery). This could potentially be more dangerous than a system that is rendered useless since user will continue to operate on the system with the assumption that all tuplespaces are correct and consistent.

2. Replication: The system is rendered completely useless in this situation and is obvious to the user that some failure has taken place.

2.1.5 What the correct behavior should have been:

1. Recovery Server (log file): A correct behavior for this implementation should have written only a single event to the recovering tuplespace and not left the log file in a duplicated state. This could have been achieved using unique identifiers for each tuple being written, or by having the the new tuplespace not multicast its events during the duration of the replication. An approach similar to globally coordinated checkpoint can be used in which a message can be multicast to the group to halt operation. Upon confirmation that the group has temporarily halted operation, recovery of the tuplespace may begin. Once complete, another multicast is sent to inform the group that operation may resume. This will lead to a consistent new tuplespace, and the halting of operation by the others ensure that it didn't miss any operations while in the process of recovering. Essentially a two-phase commit.

2. Replication The proper behavior for Replication would have performed a single operation at each of the other tuplespaces and no more. We have implemented a two-phase commit to rectify this. Each server has a log of entries. When a client connects and request an action, the server will hold a vote request and if it receives a majority of responses in the affirmative, it will proceed to multicast a vote commit message so the tuplespaces can commit the entry and perform the action, and finally the coordinator itself will commit the entry and perform the action.

2.2 Single-point of failure

The first approach introduces a singular point of failure in the naming server. If the nameserver is unavailable, there is no way for the system to proceed. At this point, clients are unable to interact with the system at all. As a result of the architecture, it is the singluar point of entry into the entire distributed platform. This can be ameliorated by abstracting away the responsibilities of a single process into that of a process group, by replicating the naming server to several well-known addresses. This would give the system some redundancy in case one of the naming servers fails.

We attempted to address this problem with approach two, with our decentralized architecture. However, we found that without some additional implementation that this was an untennable approach. At a minimum we would need to sign each message with a sequence number and an origin so that we can ignore messages that we have already seen. Without the ability to inspect and ignore messages before processing them, the decentralized microblog platform is unusable after a single message is propagated throughout the system.

2.3 Sequential Consistency

Both approaches had an issue with maintaining consistency between the data stores. There was no consensus algorithm for coordinating tuplespace operations between all of the servers. Without a mechanism for consensus, our tuplespaces quickly fall into an unsynchronized state. This issue arises from the fact that the original architecture used a single phase commit model, wherein a tuplespace would respond immediately to received events.

To address this problem, we looked into two-phase commit, where in operations are tentatively queued before being applied to the data store/state machine. This is a rather straightforward solution to implement, assuming a reliable communciation channel. The model we based our solution off of is Raft's replicated log/replicated state machine. The idea is that a single privileged node (i.e coordinator) determines the order of sequences seen by all the servers in the cluster. In the first phase, the coordinator disseminates the changes to all its followers, having them append new entries to their log. Once a majority of the servers have confirmed that they have received the event, the coordinator sends a second round of messages, telling the followers that the entry is committed. It is at this point that the individual server nodes apply the operation in the entry to their data stores.

This allows us to enforce a consistent view between all of the replicas. Because all processes in the group see the same log (i.e the same order of operations), applying all the events in log order to the tuplespace ensures that if a server is upto date on its log entries, that its state is consistent as well.

3 Proposed Implementation

After reading through the Raft Consensus Algorithm introduced by Ongaro and Ousterhout, we have decided to use it to solve some of the issues we had in our microblogging platform. As before, the underlying model is left unchanged. That is, clients should be able to transparently connect to any of the servers in the microblog platform and have their operations replicated to every other server in the cluster. The server cluster should be fault tolerant, and if the Raft Consensus Algorithm is implemented properly, the cluster should be able to tolerate (N / 2) - 1 failures. That is, if there are 5 servers in a cluster, up to 2 may fail at any given time without impacting service availability.

3.1 Raft Overview

Servers in the cluster can take on one of three states:

1. Follower
2. Candidate
3. Leader

In the normal course of operation, servers in the cluster are arranged in a master-slave relationship, wherein all log entries flow from the leader to the other servers. To be clear, clients will connect to one of the servers in the cluster, and those servers will forward operations to the Leader/Master server. There is room for implementing additional transparency at this level, such that the client can be completely ignorant of where they are connecting. Upon receipt of an event, the Leader will append the Entry to its Log, and replicate the entry to other servers in the cluster via RPC. Once a majority of servers have appended the entry to their own logs, the leader server will initiate a second round of messages, notifying its followers that the entry is committed, apply its operation to its state machine (e.g a tuplespace), and returns the result to the calling client.

3.2 Transport

We will use ZMQ REQ/REP sockets as the transport channel for internal cluster communication. Upon initialization, each server in the raft cluster binds a REP socket, allowing it to respond to incoming requests. Whenever a raft server needs to talk to a peer (e.g soliciting election votes, heartbeating as leader), we spawn a new thread for that communication with an ephemeral REQ socket.

3.2.1 RPC

We are using zmq sockets to implement the RPC pattern for internal raft cluster communication. When a server in the cluster transitions into the candidate state, they need to start an election. An election is started by the server node in question changing its state to "candidate", voting for itself, and then requests a vote all the servers in the cluster using the RequestVotes RPC. When making these RPCs, the candidate server spawns a thread that creates an ephemeral REQ socket for calling into the remote node. The remote node will be notified of the election, the address of the candidate, the candidate's proposed term, their staged entries and $\text{commit}_{\text{idx}}$. With this information the node decides whether or not to vote for the candidate.

4 Integrating tuplespaces with Vesper

We began implementing our own version of the raft consensus algorithm, loosely following the etcd implementation written in Go, when we found a Python library that supplies a raft node with an HTTP channel. With time limited, and running into substantial problems with ZMQ sockets, we decided that integrating our tuplespace/adapter code with the Raft StateMachine module was the most prudent course of action.

Vesper provides modules for initializing a Raft cluster in a known configuration (i.e view). This results in a static cluster view, because each node in the cluster needs to know about all of its peers at initialization. Without some additional implementation, it is not possible to join the configuration in the middle of its operation.

There are methods for achieving this, which are outlined in Ongaro's paper. The general idea is that we cannot simply add nodes into the configuration without a transitional period, as we must avoid a situation where in the course of migrating over to a new cluster configuration, we elect more than one leader for the same term. The main problem is that server's will not switch into the new configuration at the same time, so it is possible to have a split majority, where one leader is elected via majority from the old configuration and another leader is elected via majority from the new configuration.

To sidestep this issue, Ongaro uses a two-phase transition, whereby log entries require a joint consensus from both majorities. This transitionary period persists until we enter a safe state and enough servers have joined the new configuration. At this point the leader who is not in the new configuration will step down, and a new leader can be elected for the new configuration.

The library we used originally replicated a simple key-value store across all of its nodes. In order to connect this code with our tuplespace project, we installed a tuplespace adapter proxy inside the raft node, which allows the node to talk with our tuplespace code. Once we connected our tuplespace, we had to modify the HTTP routes to interface with our expected input and output. Once this was complete, we had successfully integrated a raft consensus mechanism ontop of our existing tuplespaces. We essentially use Vesper as a distributed commit library for imposing sequential consistency on a sequence of operations to apply to our tuplespaces.

5 Future Work

The library we used for adding raft consensus to our tuplespaces is a fair bit shy of implementing the entirety of the raft protocol. For example, it lacks dynamic cluster configuration changes, snapshotting, and log compaction. A possible approach for patching in configuration changes has been discussed above.

Snapshotting and log compaction are two issues that help solve each other. By compacting our log (i.e reducing a range of indexed log entries to a stable state) we are able to prevent the log from growing indefinitely large. We can then persist these snapshots to stable storage, and use them to restart a process from a known, stable state. This would help with latency issues across a network as we can simply send the entire state over in a request, rather than play the operations in log order as in normal operation.

In the future, I would like to branch this codebase and change the internal communication channels to use ZMQ sockets instead of HTTP. While HTTP is fine for most things, ZMQ has a few added benefits, such as being message oriented. We can also spin up multiple concurrent RPC requests over the same TCP connection which can help simplify some of the implementation. ZMQ sockets are also nice in that they don't require manual retry handling. If one end of the connection is not up, the messages will be queued rather than discarded. Additionally, there are issues with moving raft onto a wide-area network with respect to network delays. The original raft implementation will probably require some amount of modification in order to bring the election timeouts into a reasonable state on a wide-area network where congestion and other network problems are to be expected. Possible avenues of approach are to have a number of retries before declaring a link dead, and another is to use adaptive timeouts that take the estimated network delay into account.

Also, because we have naively attached the tuplespaces to this raft consensus library, there are still weakpoints in the system design. For example, if any of the processes in the tuplespace unit fail, there is no way to reliably detect this and recover. We could add some sort of watchdog process that will monitor each individual tuplespace/adapter, and if one of them fails, reinitializes it with the proper invocation. While conceptually simple, there are a lot of moving parts so this solution would require multiple iterations and stringent testing outside the scope of this course.

Furthermore, as a microblogging platform, we require a substantial amount of additional security measures, as there currently is none at all. We could extend this platform by implementing user authentication so that only authorized users can post under specific usernames. This idea can be taken further by restricting the topics that users can post to and any other number of administrative functions .

6 References

https://raft.github.io/
https://raft.github.io/raft.pdf
https://github.com/royaltm/node-zmq-raft
https://github.com/etcd-io/etcd
https://github.com/Oaklight/Vesper
http://zguide.zeromq.org/page:all
http://blog.pythonisito.com/2012/08/distributed-systems-with-zeromq.html
https://augustl.com/blog/2013/zeromq_instead_of_http/
https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/