Distributed Databases Project Report

Joseph Lewis–lewis285

Problem

This project aimed to implement a distributed three-phase commit protocol in the go programming language. The project assumes a fail-stop model of node failures and no network partitioning. This is primarily because the protocol isn't tied to any one technology. It could synchronize databases, data-structures, or even text-files.

Methods

I wrote the project in the `go` language. `go` was designed to be used for networked systems so the communication overhead is low, it's a modern language so development was fast and it's compiled so speed isn't an issue.

Components

The project is broken down into four major components:

1. "checkup"
2. "cohort"
3. "storage"
4. "threephase"

"checkup"

The checkup module provides a timeout based liveness check on all the registered nodes in the system. In our case, we just use a successful TCP handshake as being valid for liveness. Once started, the checkup daemon will try to connect to each node over an interval with a given timeout. Internally, is stores a map of which nodes are alive or dead.

"cohort"

The cohort module contains two sub-components:

- RWModes
- Cohort

An **RWMode** is the read and write mode of the system. These modes tell you how many nodes you need in order to perform a create, read, update or delete. Currently, the system has Read One Write All and Read Majority Write Majority implemented. The table below shows how many nodes are needed for each operation:

Mode	Create	Read	Update	Delete
ROWA	n	1	n	n
RMWM	1/2n	1/2n	n	n
RAWO	1	n	n	n

ROWA and RMWM could have better update and delete performance if we could guarantee that the underlying database could tell us where the data was or if it was versionable. As it is, the system has a minimal interface into the data so it can be more general-purpose.

A **cohort** combines an `RWMode` with a `updown` daemon. When the three-phase-commit protocol needs to perform an operation, it can ask the `cohort` for a list of the right number of alive nodes it can contact. For speed, the current node is always added into the list of nodes. If there are not enough nodes alive to perform an operation cohort returns an error which will abort the request.

Storage

The `Storage` interface represents the data we want to use the 3PC system on. It has the following read-only methods:

- Read – Reads data from the data-structure
- Merge – Merges multiple reads
- Stats – Generates a human-readable report about the data

A storage object must also implement three functions that 3PC will use:

- Prepare – Prepares a modification to the data-structure
- Abort – un-prepares a modification to the data-structure
- Commit – Modifies the data-structure

For example, let's say we had a distributed key/value store. Prepare would receive the key, value and a transaction identifier. If this store already had the key it would return an error, otherwise it would lock the key and wait for commit or abort. Commit or abort moves the store out of the prepared state and into a modified state or flushes the change.

threephase

This module contains the meat of the three phase commit system. On a high level the 3PC algorithm needs to have access to:

1. A list of alive and dead hosts
2. A stream of incoming messages
3. A place to put outgoing messages
4. An internal list of messages

Number 1 we have taken care of through the `checkup` and `cohort` subsystems. Number 2 (the stream of incoming messages) we defer to the user. They can decide what messages to send us by calling any of the following functions:

```
InitializeTransaction(transaction []byte) (ok bool)
Abort(transactionID string) (ok bool)
DoCommit(transactionID string) (ok bool)
PreCommit(transactionID string) (ok bool)
CheckCommit(transactionID string) (didcommit bool)
```

The demonstration software attaches these calls to HTTP endpoints to create a RESTful server.

The software also needs to provide a set of callbacks so 3PC can send the right messages off. When called, they'll make requests to invoke the methods of the same name listed above on the destination machine:

```
InitializeTransaction(tx []byte, destination string) (ok bool, err error)
Abort(transactionID []byte,      destination string) (ok bool, err error)
DoCommit(transactionID []byte,   destination string) (ok bool, err error)
PreCommit(transactionID []byte,  destination string) (ok bool, err error)
CheckCommit(transactionID []byte, destination string) (didcommit bool, err error)
```

Finally, `threephase` needs an internal list of messages. In the software, each transaction is an instance of a `ThreePhaseTransaction`:

```
type ThreePhaseTransaction struct {
    Peers         []string
    Data          string
    TransactionID string
    status        Phase
}
```

Peers lists all nodes participating in the transaction. Data is a string representing the request. TransactionID is a globally unique ID. By default, our system uses the current time-stamp in nanoseconds converted to a string. Status is the current phase of the transaction this is one of {PhaseUncertain, PhasePrepared, PhaseCommitted, PhaseAborted}.

As the methods above are called from the outside, the status of each transaction will be updated. Whenever a status change occurs, it kicks off a routine that will do the automatic "cleanup" of that state:

Event	Cleanup Protocol
Init	Termination protocol
Precommit	Auto-commit
Commit	Remove transaction from memory
Abort	Remove transaction from memory

If the state has changed since the daemon routine was spawned, the routine stops without changing anything. This is okay to do in `go` because it can easily handle 100,000+ concurrent routines.

Data

I spawned three instances of the demonstration replicated logging system and ran a benchmarking software that made repeated requests. They range from 3-5ms each for full replication:

Number of Threads	Requests/sec
1	194
2	292
4	357
6	338
8	336

An organic system would likely have slightly more latency than hosts all running on a single machine. However, the production system would also likely use a better RPC method than establishing new TCP sessions and making HTTP calls each time.

Experiences

Overall, I enjoyed building this system in the framework I chose. goroutines made cleanup much easier than managing threads. The communication primitives and `select` statements are very conducive to building systems that require safety, performance and prevention of deadlocks.

Observations

The performance this system provides is decent, even if it might not be appropriate for Amazon. Future systems based on this work will probably want to multiplex all the communication handlers into a single callback to further abstract away the 3PC system.

It would be prudent to keep a single connection open with between each pair of nodes and send data through those existing ones. The benchmarks hitting a wall seemed to imply that either my loopback device was too slow or it was the HTTP connections causing the overhead. Especially for short messages, a full HTTP session for each part of the protocol (init, precommit, commit, abort, recover) begins to be the dominating factor in communication.

Currently the cohort sub-system always puts the coordinator's machine into groups for performing transactions and chooses the rest of the cohort at random. A larger system could take into account other metrics to decide which hosts are included. In the first week of the class we talked about the economics of distributed systems, all of that could factor in. Which hosts have the highest bandwidth? Which have the most storage? Which are accessed the most? Which are constantly crashing? You could even put business rules on top like, "data from Europe can't be stored on US servers."

Correctness testing the protocol was difficult. Abstracting everything away behind interfaces works to an extent, but all abstractions are leaky. I ended up emulating networking and responses through the interfaces I created rather than actually manipulating network traffic because it was more reproducible. I know that CERIAS has some kind of network simulator for testing distributed systems although it didn't seem to be any easier to use than abstracting interfaces.

It's clear that 3PC is best used for small, close together systems due to the message overhead. Google really did a good job here with GFS. The chunkservers are replicated using 2PC or 3PC and there are only a few of them, three by Google's claims, per chunk. Maintaining a read-majority-write-majority system definitely isn't feasible when you scale, but neither is a read all write one because of dependability problems. Introducing a hierarchy is a fantastic way to solve the overhead vs durability problem.