6.824 Project: GoAway

Jessica Kenney, Andres Perez, Miles Steele

https://github.com/anpere/goaway

May 12, 2016

1 Introduction

GoAway is a Python library that allows users to easily spawn processes on remote machines in a cluster. Users who have programmed in Go will recognize some of the library semantics. GoAway also provides users with the ability to allocate distributed shared memory among the machines in the cluster.

In designing GoAway, we aimed to provide a system that is easy to use and provides a variety of consistency models that the programmer can choose from. We assume a fully reliable network and servers, and do not address durability at all, which inherently limits the scalability of our system.

1.1 Defined Terms

We use the following terms throughout the paper and code:

- **Spawner** The machine that runs the GoAway init routine, and spawns the remote servers. In most use-cases this is the user's computer.
- **Command Server** The HTTP server running on every computer which receives "commands" from other nodes in the cluster.
- **DataStore** An abstract notion of a place where objects are stored. In a single GoAway cluster, there will be one datastore for each consistency model. GoAway uses DataStore handles to implement DataStores.
- **Object** An abstract notion of an object that holds arbitrary data which is managed by the datastore corresponding to that object.
- **Value** The data held in each object. Users can get and set Values.
- **DataStoreHandle** Only used internally in the library, each goaway process talks to its local DataStoreHandle. The DataStoreHandle may talk to other GoAway processes via remote RPC.
- **ObjectHandle** This is the object that is returned to the client. It provides an interface to get and set values, as well as object synchronization methods if its consistency model requires them.

1.2 Use case

Figure 1 shows a high level overview of the system. Typically the user will sit at the spawner machine and write application code. When the user decides to run the application, the GoAway library will allow them to do two things. First it will launch GoAway processes on the spawner machine and on the remote machine. Second, the GoAway library provides an interface to allow the user to spawn GoAway routines on the GoAway processes on the remote machines. Each remote machine can communicate with each other using distributed shared memory, which the GoAway library supplies an interface for.

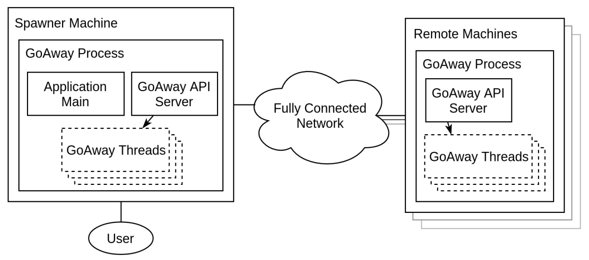

Figure 1: System diagram.

2 API

Figure 2 lists the delegation of responsibilities between the user and the GoAway library. We aimed to minimize the amount of work the user did, while minimizing the complexity of GoAway. With the exception of updates to the GoAway library, all of the actions performed by users are performed at most once on each machine. It would be too much work for users to have to upload application code on remote machines every single time the application code changes. The GoAway library provides functionality to upload application code, and make remote servers respond to function calls.

Task	Delegation
Provision remote servers	user
setup full mesh network amongst peers	user
Install GoAway library and dependencies on all machines	user
Install application dependencies on all machines	user
Upload latest application	GoAway
Start GoAway servers	GoAway

Figure 2: Table of responsibilities

2.1 GoAway library calls

These are the following methods that users can call:

- **init(config_file)** Prepares the remote servers to listen to commands issued by the user's computer. Uses the config_file to know what application code to send, and the ip addresses of the available machines.
- **goaway(function, *args, **kwargs)** Calls function on an available machine with the listed arguments. The function must be defined in the global namespace of the application, and it can use globally defined variables including shared objects.
- **StrictCentralized(name)** Returns a handle to a shared object. The application can read and write values on the object, which are shared with other remote processes according to the Strict Centralized consistency model described in section 3.1.1. The name is used internally to uniquely identify the object.
- **LinFastRead(name)** Returns a handle to a shared object whose data is propagated according to to the Local-Read Linearizable consistency model described in section 3.1.2.
- **Weak(name)** Returns a handle to a shared object whose data is propagated according to to the Weak consistency model described in section 3.1.3.

```
import goaway
lock = goaway.Lock("lock")
s = goaway.StrictCentralized("s")

def increase(value):
    with lock:
        s.counter += value
if __name__ == "__main__":
    goaway.init(config_file_path)
    for i in available_servers:
        goaway.goaway(increase, 1)
```

Figure 3: Sample code

- **UpdateOnRelease(name)** Returns a handle to a shared object whose data is propagated according to to the Release consistency model described in section 3.1.4.
- **Lock(name)** returns a lock that is uniquely identified across processes by its name. Provides a blocking acquire and a non-blocking release. The imprementation is described in more detail in section 3.1.5.

Figure 3 shows a distributed counter that uses the GoAway library. When init is called, GoAway sends the application code below to each machine in the user's cluster. When the user calls goaway with the function increase, increase is called on a remote machine. The first time the machine receives a call to increase, it loads the application code, which then creates instances of s and lock.

3 Implementation

The users of GoAway can view the system abstractly as shown in Figure 4. The user can sit at their computer and spawn GoAway routines on separate machines that all operate on the same objects in the shared datastores. Each access to these objects satisfy the semantics of the datastore they live in.

In reality, the shared memory between machines must exist somewhere. Figure 5 shows how that is implemented in our system. For each abstract datastore, every machine has a copy of a datastorehandle which is used to coordinate operations on the objecthandles contained in that datastorehandle. Our objecthandles intercept the gets and sets on the python objects, and properly coordinate with the datastorehandles on other machines to properly satisfy the semantics of that datastore.

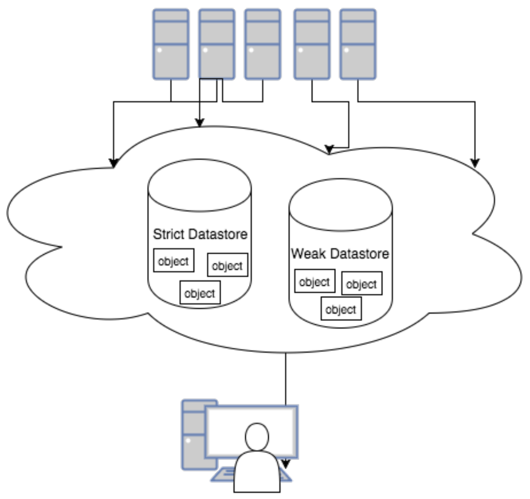

Figure 4: Abstract view of shared memory.

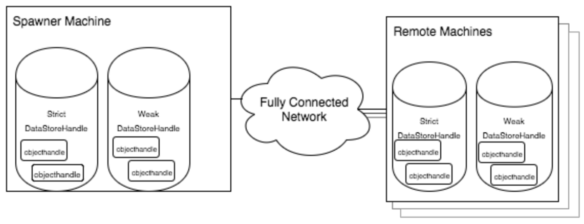

Figure 5: Concrete view of shared memory.

3.1 Data Stores

We implemented four datastores[1] with different speed and consistency properties. Each machine has a DataStoreHandle to interface with the datastore.

3.1.1 Strict Centralized

The StrictCentralized datastore is stores all instances of StrictCentralized objects. It is a simple single-master service where all values are stored on one machine. In this case the first machine in the config list that is not the spawner machine. All gets and puts are performed as blocking RPCs to the master server.

This datastore has the worst performance, but also has the highest consistency.

3.1.2 Local-Read Linearizable

The local-read linearizable datastore is available through the LinFastRead constructor. It provides per-value linearizability within each object and it optimized for reads. So for an GoAway object x created by LinFastRead, x.a is one value and x.b is another value. This datastore is best used for data which an application frequently reads but only very infrequently writes. It can also be used in conjunction with a lock to provide transactional guarantees.

[1]Note ReleaseConsistency doesn't quite work yet, and needs to be fixed. Nevertheless we thought it would be useful to discuss the idea behind it.

Every machine maintains a local cache of all shared data in the store. Whenever the application attempts a read, the read can be performed immediately locally with no RPCs, and is guaranteed to obtain the latest value.

Writes are slower and involve a two-phase locking commit. In step 1, the expanding phase, the writer locks the unit on all servers in a global ordering (provided by the config file when GoAway was initialized). Using the same lock order every time guarantees that deadlock never occurs. The writer waits for each of the servers to respond to the unit lock request before continuing. After all locks have been acquired, step 2, the shrinking phase begins. The writer sends an update message to each server in the same order. This message contains the new value to install. Once a server receives update message it updates its local store and unlocks the object.

Reads occur even when objects are “locked” and this does not violate linearizability because the write does not return until all updates have been acknowledged. Since there is no situation in which writes are ever rolled back (even before the second phase of the commit), it is always safe to read immediately.

3.1.3 Weakly Consistent

The weakly consistent datastore maintains a copy of the data on each process. On reads, the local value is returned immediately. On writes, the value is written locally and RPCs are sent asynchronously to every other server in the cluster. When the datastore receives a write from another server it installs it immediately.

The Weak datastore also provides a per-object synchronization operation, which blocks until all outgoing write RPCs have finished. So if a process P1 writes x=1 and then synchronizes, it is sure that at that point any other process would read x=1.

3.1.4 Release-Consistent

The release-consistent datastore provides release consistency semantics. An object that exists in a release-consistent datastore has a lock which users can acquire and release. Whenever a user releases the lock associated with an object, the changes that were made to that object are propagated to the rest of the machines.

Changes to the object are applied to a local cache in the datastore, and logged in a buffer. When the lock has been released, these changes are sent to the release-consistent datastores on the other machines.

3.1.5 Locks

The Lock is a datatype provided by the GoAway library. It isn't a datastore, but it acts similarly to one. One server is designated as the master lockserver. Whenever a process tries to acquire the lock, it blocks and sends acquire-request RPCs to the master lockserver, who replies with true or false depending on whether the acquire was successful. The lock itself resends the acquire-request RPC until it's successful. The lock release asynchronously sends a release RPC to the master lockserver. This lock is not reentrant.

4 Limitations

Each of these limitations reduces the ease of use of goaway in some way, but we could probably overcome most of these limitations in a future version.

- Only immutable, serializable values can be safely stored in goaway objects. Mutating something stored in a goaway object would not be propagated across processes because goaway wouldn't know about the change.
- Functions which are sent via goaway must be declared in the global scope of the application in order to be runnable by remote machines. Lambdas (anonymous functions) are not supported.
- The spawner machine must be able to run a public HTTP server that all other machines in the cluster can access, so it can't be behind a NAT.