Home Exam 2

Victor Nascimento Bakke victonba@ifi.uio.no

The Framework

Emerald does not (to my knowledge) provide any tools or facilities for reflection, or inspecting an arbitrary object and discovering its methods at runtime. As such, it is impossible to create a framework that will accept any arbitrary object to propagate its state through Primary Copy Replication (PCR). Since we need to know, statically, how to update the contained object, we require a total of three operations to be available on whatever is contained in the PCR framework: `cloneMe` for cloning the contained objects, `addToState` to add new content to the contained object and `removeFromState` for removing content from the object. This is a rather rudimentary implementation, but it should work for our use cases. The objects in the framework are kept in either a `Primary` or a `Replica`. A `Primary` has zero or more associated `Replica`s, while a `Replica` has one, and only one, `Primary`. These two implementations of the PCR containers are made ambiguous through the `State` type, allowing us to easily have both a `Primary` and one or more `Replica`s in one `Array`, for example.

`State` has four operations: `getState`, `addToState`, `removeFromState` and `halt`. `getState` returns the internal state of the `State` in question. `addToState` and `removeFromState` behaves differently depending on whether the underlying container is a `Primary` or a `Replica`. `halt` is used to stop running processes.

`Primary` calls `addToState` and `removeFromState` on the contained object, and queues either an "add" or "remove" type of update to be propagated to the `Replica`s. `Replica`s instead call `addToState` and `removeFromState` on its parent `Primary`, ensuring the new state is propagated by `Primary` at some later point in time.

`Primary` has a queue of updates that shall be propagated to its `Replica`s and a process which loops through this list and sends these updates periodically. Thus, utilizing the Observer Design Pattern, the replicas are updated by their `Primary` whenever `Primary` sees an update.

The `State`'s `halt` operation is only used to stop `Primary`'s process, and have the program exit properly. `Replica`s delegate to their parent's `halt` operation. It is rather unnecessary, since the program has to be halted with `SIGINT` (`CTRL-C`) when `emx` is called with the `-R` option regardless.

The basis of these assumptions stem from the assignment text as well as the interpretation discussed in the question posed on Piazza.

Name Server

Output file: `nameserveroutput.txt`

A name server has four operations: `lookup`, `add`, `addToState` and `removeFromState`. `add` is simply used to add new entries to the name server. `lookup`, as required by the assignment, simply returns the `NameServerObject` in its collection whose name matches the passed string. `addToState` is a synonym for `add`, while `removeFromState` removes the matching entry from the `NameServer`'s list of entries. Both of these operations are required to be present by the PCR framework.

A `Client` interacts with a given name server , and periodically looks up a specific entry from the list of all entries. After between 6 to 10 seconds, it adds an entry to the name server, and updates the state. Once the test is complete, the `Client`s' fetched results are printed to the console.

Time Server

Output file: `timeserveroutput.txt`

If the `Primary` should be able to update the time of day on its own, the PCR framework would have to support receiving updates from the contained objects. The contained time server objects that are being contained in the `State`s would have to be treated differently, since only one of them would be a `Primary`. Alternatively, the `Primary` could call a method on its contained object, which is a whole 'nother bag of complicated and does not make for a sensible framework for these purposes.

Thus, updates will still come from the outside, with a `TimeSetter` updating the time periodically, and `TimeGetter`s asking the replicas for the current time of day. These `TimeGetter`s collect the times they've gathered so that we can list them up in the end.

`TimeServer` is the object being contained in the framework, and has four operations: `getTimeOfDay`, `setTimeOfDay`, `addToState` and `removeFromState`. `getTimeOfDay` simply returns the contained time in the `TimeServer`. `setTimeOfDay` overwrites the contained time with the new time passed to the operation. `addToState` is an alias for `setTimeOfDay`, while `removeFromState` is a `noop`, since removing any sort of item from the state does not make much sense in this case, but its presence is still required by the framework.

`TimeGetter` are objects that simply fetch the current time of day, and store them in a collection. After the test is complete, we print the contents of the `TimeGetter`s' collected times to the console.

Building and running

Using the script `run` in the root of this project, building and running the two tests is rather easy:

Building

To build either the name server test or the time server test:

```
./run build <nameserver|timeserver>
```

i.e. `./make build nameserver` would build the name server.

Running

To start the tests:

```
./run start <nameserver|timeserver> [options]
```

Available options are `-U` and `-R`, which are passed directly to `emx` as-is.

Alternatives

Instead of requiring `addToState` and `removeFromState`, I could've opted for a more generic `updateState[String, itemType]` operation where the first `String` argument would describe the kind of update to take place. Personally, I did not see the benefit of this approach compared to using statically checked operations.

PlanetLab setup

- planet1.elte.hu
- mars.planetlab.haw-hamburg.de
- ple41.planet-lab.eu
- csl12.openspace.nl