CSE 4/586: Project 2

Murat Demirbas

[2016-11-14 Mon]

1 Replicated Storage with Chain Replication

In this phase of the project, we will use *Chain Replication* to ensure *persistence* and *consistency* of the objects stored. Before performing the write, the client reads from the *tail* of chain to learn the highest versioned write completed. The client then increments the version number, and performs the write to the *head* of chain with this version number. Since a node failure may lead to a request being lost (be it a *read* request, or *update* request), the client needs to repeat the request until a response is returned from the tail. The client cannot start a new request before it receives a reply to its earlier request.

A *configurator* process p (an infallible process which would in practice be implemented as a Paxos cluster) will be used for configuring the chain. One responsibility of the configurator is to remove a down node from the chain, and the other is to add a new node to the tail of the chain if the length of the chain is less than 3. The chain is initially empty, and the configurator populates the chain using the "add a new node to the tail" action.

A storage node can crush (provided that *FAILNUM* is not exceeded), and recover at any time. For simplicity we assume the client writes to only one item, so we omit the key part of the key-value pair item, and model the database *db* at each node to consist of one item. The newer version of the item will writeover the old version.

In this phase of the project, the storage system performs *server-side routing*, and modeling of the system is done using message passing instead of shared memory. Once a storage node receives a new request in its message-box *msg*, it will consult the configurator to figure out its successor in the chain and propagate the request to its successor. If the storage node is the *tail* it will send a reply to the client. An *update* request modifies the *db* with the newer version of the item. A *read* request does not change the *db*.

1.1 Write a PlusCal program to represent this algorithm.

Use the template we provide as your starting point, and fill in the redacted parts. Use the toolkit to translate your code to TLA+ and model-check for correctness.

1.2 Model-check safety properties with TLA+

- Write two invariant properties to capture the single-copy consistency property of the storage protocol. Single-copy consistency means the *N=3* storage system appear to outside as if it is a single virtual infallible node, even though upto *FAILNUM* of the storage nodes can fail. In other words, this implies that the highest version number result returned by the *tail* of the chain should match the item stored by the most recent completed write operation on the system.

  - For simplicity the client only implements the writes to the system. But you can still check that consistency is satisfied without having to implement the read operation by leveraging the way the client stores items to the storage system.
  - As the second invariant property, check that *db.ver* is non-increasing as we traverse the chain from head to tail.

- Model check with *C=2* clients to find out which of the above two properties still hold.

- Write in the comments section, after the `"================"` line, your findings/observations about this version of the protocol *Voldchain* with the *project1* version of the protocol. How do they compare? Is Voldchain capable of tolerating more failures with less number of nodes? How do you explain the difference? What is the analog of write quorum, and read quorum in Voldchain? Does the previous relation between quorums and FAILNUM still hold? What are your observations from *C=2* clients?

2 Voldchain

MODULE *vchain3*

Chain replicated storage protocol. Serverside routing

EXTENDS *Integers*, *Sequences*, *FiniteSets*, *TLC*
CONSTANTS N, C, $STOP$, $FAILNUM$
ASSUME $N - FAILNUM \geq 1 \land STOP < 5 \land 0 \leq FAILNUM \land FAILNUM \leq 2$
$Nodes \triangleq 1 \,..\, N$
$Clients \triangleq N+1 \,..\, N+C$ * should give different *ID* space to Client
$Procs \triangleq 1 \,..\, N+C$
$Configurator \triangleq N+C+1$ * *Configurator* is unfallable

```
--algorithm voldchain
{
  variable FailNum = FAILNUM,
           msg = [j ∈ Procs ↦ ⟨⟩],   \* Each process has an inbox
           up = [n ∈ Nodes ↦ TRUE],   \* Initially all nodes are up
           db =   \* db is single record only
              [n ∈ Nodes ↦ [ver ↦ −1, val ↦ −1, cli ↦ −1]];
           chain = ⟨⟩;   \* chain is a sequence, initially empty

  define
  { UpNodes ≜ {n ∈ Nodes : up[n] = TRUE}
    InChain(s) ≜
    ChainNodes ≜
    FreeUpNode ≜
    GetIndex(s) ≜   \* Assumes InChain(s), returns index of s in chain

    GetNext(s) ≜   \* Assumes InChain(s), returns successor of s in chain

  }

  fair process ( c ∈ Clients )   \* Client process
  variable cntr = 0, hver = −1;
  {
   C0: await (Len(chain) > 0);   \* start requests only after chain is constructed
   CL: while ( cntr ≤ STOP ) {
CLR:

CLW:

    }
  }
```

```
fair process ( n ∈ Nodes )   \* Storage node
{
ND: while ( TRUE ) {
      either
      NM: {   \* react to message
                [redacted]
       } or
    NDF: {
           if ( FailNum > 0 ∧ up[self] = TRUE ) {   \* Storage node can fail, db & c
             up[self] := FALSE ;
             FailNum := FailNum − 1 ; }
           else if ( up[self] = FALSE ) {   \* Or recover as new node
             up[self] := TRUE ;
             msg[self] := ⟨⟩ ;
             FailNum := FailNum + 1 ; }
           }
    }
 }
fair process ( p = Configurator )   \* Maintain the chain
{
P: while ( TRUE ) {
      [redacted]
    }
 }
}
```

3 Submission

Your TLA+ file should be named *voldchain.tla* . Your model's name should be the default name *Model_1* (do not name your model file differently).

Generate a pdf print of your TLA+ program using the "Produce Pdf version" from the TLA+ menu. (This will get included in your submission as it is created under the ".toolbox" directory.)

Now create a zip file from the ".tla" file and the corresponding ".toolbox" directory. **Name the zipfile as: proj2.zip**

Not following these directions will cause you to lose points.

You will use the submit command (*submit_cse*486 or *submit_cse*586 respectively) to submit your work. The submit command instructions are here: `https://wiki.cse.buffalo.edu/services/content/submit-script`