Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 47 November 22, 1993

Replication Techniques

This handout presents specifications and implementations for a variety of replication techniques. We start with two specifications of a "strongly consistent" replicated service, which looks like a single copy to its clients (except that some client requests can fail). (The second specification constrains the failure behavior more than the first.) Then we give two implementations, one based on primary copy and the other based on voting. Finally, we give a specification of a "loosely consistent" service, which has a much weaker specification than would be expected from a non-replicated service.

1 Specifications of Strongly Consistent Services

A strongly consistent service executes actions just like a non-replicated service: each action is executed at most once, and all clients see the same sequence of actions. However, the response to a client's request for an action can also be that the action "failed"; in this case, the client does not know whether or not the action was actually done. The client may be able to figure out whether or not it was done by executing more actions, but the failed response gives no information.

The first specification places no constraints on the timing of failed actions. If a client requests an action and receives a "failed" response, the action may be performed at any later time. In addition, a "failed" response can be generated at any time.

The second specification further constrains failed responses so that they can be generated only if the system fails (or is recovering from a failure) during the execution of an action. However, actions with failed responses can still be performed at any later time.

In practice, some constraints on when failed actions are performed would be desirable, but it seems hard to write a general specification of such constraints that applies to a wide range of implementations. For example, a client might like to be guaranteed that all actions, including failed actions, are done in the order in which the client requests them. Or, the client might like the same kind of ordering guarantee, but covering all clients rather than each individual one separately.

Here is the first specification, which allows "failed" responses at any time:

```
MODULE AMO [
   V,                                           % Value
   S WITH { s0: () -> S },                      % State
   A WITH { meaning: A -> S -> (V,S) }          % Action
   ] =

VAR s := S$s0()                                 % State of service
VAR pending: SET A := {}                        % Failed actions to be done.

% Either do the action or raise failed.
APROC Do (a) -> V RAISES {failed} =
  << VAR v,s' | (v,s') = a.meaning(s) => s := s'; RET v
    [] pending := pending ++ {a}; RAISE failed >>

% Do a pending failed action.
THREAD DoPending() =
  << VAR a | a IN pending =>
      VAR v,s' | (v,s') = a.meaning(s) => s := s'; pending := pending - {a} >>

% Discard a pending failed action.
THREAD LosePending() =
  << VAR a | a IN pending =>
      pending := pending - {a} >>

END AMO
```

Here is the second specification. Intuitively, we would like a response of “failed” to occur only if the service fails (a crash or a network failure) sometime during the execution of the action, or if the action is requested while the system is recovering from a failure. To capture the first constraint, the specification needs to model the duration of an operation. This is why the `Do` procedure is specified as a sequence of two separate atomic actions.

```
MODULE AMO2 [
   V,                                           % Value
   S WITH { s0: () -> S },                      % State
   A WITH { meaning: A -> S -> (V,S) }          % Action
   ] =

VAR s := S$s0()                                 % State of service
VAR pending: SET A := {}                        % Failed actions to be done.
VAR down: BOOL := false                         % Flag: true when system has
                                                % failed and not finished
                                                % recovering
```

```
% Either do the action or raise failed.
% failed can be raised only if the system is down sometime during the
% execution of the operation.
APROC Do (a) -> V RAISES {failed} =
  VAR failure: BOOL |
  << failure := down >>;
  << VAR v,s' | (v,s') = a.meaning(s) => s := s'; RET v
    [] failure => pending := pending ++ {a}; RAISE failed >>

% Do a pending failed action.
THREAD DoPending() =
  << VAR a | a IN pending =>
      VAR v,s' | (v,s') = a.meaning(s) => s := s'; pending := pending - {a} >>

% Discard a pending failed action.
THREAD LosePending() =
  << VAR a | a IN pending =>
      pending := pending - {a} >>

% Appears to happen whenever a node crashes or the network fails.
THREAD Fail() = << down := true >>; << down := false >>

END AMO2
```

2 Implementations

There are two general ways of implementing a replicated service: primary copy (also known as master-slave, or primary-backup), and voting (also known as quorum consensus). Here we sketch the basic ideas of each.

2.1 Primary Copy

The primary copy algorithm we sketch here is based on one invented by Barbara Liskov and Brian Oki, and uses the Paxos consensus algorithm. When things are working well, the client sends requests to the replica that is currently the primary; that replica uses Paxos to reach consensus among all the replicas about the index to assign to the requested action, and then responds to the client. An index i will not be assigned to an action unless all prior indices have been assigned to actions, and no later indices have been assigned. Furthermore, the primary will know the actions associated with all earlier indices, so it can compute the result to return to the client.

```
MODULE PrimaryCopy [
   C,                                          % Client names
   R,                                          % Replica names
   V,                                          % Value
   S WITH { s0: () -> S },                     % State
   A WITH { meaning: A -> S -> (V,S) }         % Action
   ] =

TYPE
   K  = ENUM[req,resp]                         % Kind of message
   D  = (A + V)                                % Data in message
   ID = INT                                    % Ids for messages
   P  = (R + C)                                % All process names
   M  = [sp: P,                                % Message: sender name,
         rp: P,                                %        receiver name,
         id,                                   %        uid,
         k,                                    %        kind,
         d]                                    %        data
   STATE = ENUM[idle,trying]                   % State of primary replica
   AID = [A, ID]                               % Action plus uid for consensus

VAR
   actions: INT -> Consensus[AID]              % A separate consensus instance
                                               % for each action index; outcome
                                               % records the agreed-upon action
                                               % and its id.  Consensus is run
                                               % by the processes in R.
   msgs: SET M  := {}                          % Messages in transit
   used: SET ID := {}                          % Uids used so far

PROC Do(a,c) -> V RAISES {failed} =
   VAR primary: R,             % guess the current primary
       id | ~(id IN used) =>   % and choose a new uid
   << used := used ++ id >>;
   Send(c,primary,id,req,a);
   VAR d := Receive(primary,c,id,resp) |
        d IS V => RET d
        [*] RAISE failed
   [] RAISE failed             % if timeout waiting for response
```

```
THREAD Actions(r) =
   VAR  % Only primary has state in this module.
     c,            % client to respond to
     id,           % id of latest request
     a,            % latest requested action
     i: INT,       % index to try
     state         % state of primary
  |
   DO BEGIN <<    % Pick an enabled action and do it.

   % Primary: receive request when idle
   VAR c',id',d | d = Receive(c',r,id',req) /\ state = idle /\ d IS A =>
        c := c'; id := id'; a := d;   % record request
        i := TryIndex(a,id);          % Find index to try and start consensus
        state := trying

   % Primary: check outcome of current consensus
   state = trying /\ actions(i).Outcome() IS AID =>
      VAR aid := actions(i).Outcome() |
        aid.id=id => % success; return value and become idle
          Send(r,c,id,resp,Value(i));
          state := idle
      [*] % another action got this index; try again
          i := TryIndex(a,id)

   % Crash
   state := idle

   >> END OD

APROC TryIndex(a,id) -> INT =
  << VAR i | i > 0
             /\ actions(i).Outcome()=nil
             /\ actions(i-1).Outcome() IS AID =>
        actions(i).Allow(AID{a:=a,id:=id}); % start consensus
        RET i >>

% Compute value returned by i'th action; assumes that outcomes
% are available for all indices less than or equal to i.
APROC Value(i:INT) -> V = RET ValueInState(1,i,S$s0())
```

```
% Apply actions from lo to hi to s and return value
APROC ValueInState(lo:INT,hi:INT,s) -> V =
  << actions(lo).Outcome() IS AID =>
        VAR v,s' | (v,s') = actions(lo).Outcome().a.meaning(s);
            lo < hi => RET ValueInState(lo+1,hi,s')
        [*] RET v >>

APROC Send(p1,p2,id,k,d) =
   << msgs := msgs ++ M{sp:=p1,rp:=p2,id:=id,k:=k,d:=d} >>
APROC Receive(p1,p2,id,k) -> D =
   << VAR m | m IN msgs /\ m.sp=p1 /\ m.rp=p2 /\ m.id=id /\ m.k=k
       => msgs := msgs - {m}; RET m.d >>
THREAD LoseMessage() =
   << VAR m | m IN msgs => msgs := msgs - {m} >>

END PrimaryCopy
```

2.2 Voting

The voting algorithm sketched here is based on one invented by Dave Gifford. We abstract away from the details of communication and locking.

```
MODULE Voting [
   R,                                          % Replica names
   V,                                          % Value
   S WITH { s0: () -> S },                     % State
   A WITH { meaning: A -> S -> (V,S) }         % Action
   ] =

VAR
  states: R -> S   := (* -> S$s0())          % States of replicas
  vnum:   R -> INT := (* -> 0)               % Version numbers of replicas
  rqs:    SET SET R                          % Read quorums
  wqs:    SET SET R                          % Write quorums
    % rqs and wqs are initialized in the body at the end of the module
```

```
APROC Do(a) -> V =
  << ReadOnly(a) => % Read, not update
     VAR rq: SET R | rq IN rqs =>
        VAR r |
           r IN rq /\ vnum(r) = {r' | r' IN rq | vnum(r')}.max =>
        VAR v,s | (v,s) = a.meaning(states(r)) => RET v
  [*] % Update action
     VAR wq: SET R | wq IN wqs =>
        VAR r, vn: INT |
           r IN wq /\ vnum(r) = {r' | r' IN wq | vnum(r')}.max =>
        vn := vnum(r) + 1;  % new version number
        VAR v,s | (v,s) = a.meaning(states(r)) |
        DO VAR r' | r' IN wq /\ vnum(r') < vn =>
            vnum(r') := vn; states(r') := s OD;
        RET v
  >>

FUNC ReadOnly(a) -> BOOL =
  RET (ALL s, s', v | (v,s')=a.meaning(s) ==> s=s')

% chooses sets of read and write quorums such that every read quorum
% intersects every write quorum, and every write quorum intersects
% every other write quorum.
APROC Quorums () -> (SET SET R, SET SET R) =
  << VAR rqs: SET SET R, wqs: SET SET R |
      (ALL rq: SET R, wq: SET R, wq': SET R |
         rq IN rqs /\ wq IN wqs /\ wq' IN wqs ==>
              (rq*wq).size > 0 /\ (wq*wq').size > 0)
     => RET (rqs,wqs)

BEGIN
  (rqs, wqs) := Quorums()  % Initialize read and write quorums

END Voting
```

3 Specifications of Loosely Consistent Services

Some services have availability and response time constraints that make it impossible to maintain the illusion that there is a single copy. Instead, each operation is initially processed at one replica, and the replicas "gossip" in the background to keep each other up to date about the updates that have been performed. Such strategies are used in name services, for distributing information such as password files, and for maintaining system binaries.

Propagating updates in the background means that when an action is processed, the replica processing it might not know about some earlier actions. This is reflected in the specification below by allowing an action to be processed using any subset of the earlier actions to determine the response to the action. Such behavior is possible (though unlikely) in systems such as Grapevine. A more useful specification would limit the nondeterminism by requiring an action's response to include the effects of all actions executed before a certain time, which the system would attempt to keep relatively recent. An implementation providing this stronger guarantee was developed by Butler Lampson and others at DEC in their design of their "global name service".

```
MODULE LooseReplication [
   V,                                          % Value
   S WITH { s0: () -> S },                     % State
   A WITH { meaning: A -> S -> (V,S) }         % Action
   ] =

VAR sa: SET A           % State is just set of updates performed; we assume
                        % that the same update is never requested twice

APROC Do(a) -> V =
  << VAR sa': SET A, v | sa' <= sa =>
      v := DoActions(sa',a);
      sa := sa ++ a >>

% Do actions in sa' in some order, then return result of a.
APROC DoActions(sa': SET A, a) -> V =
  << VAR s := S$s0() |
     DO VAR a',v,s' | a' IN sa' /\ (v,s') = a'.meaning(s) =>
       s := s'; sa' := sa' - {a'} OD;
     VAR v,s' | (v,s') = a.meaning(s) =>
       RET v >>

END LooseReplication
```