Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 28 May 5, 1997

Replication

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.

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
    ] EXPORT Do =

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

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

THREAD DoPending() =                            % Do a pending failed a
    DO << VAR a :IN pending | pending := pending - {a}; s := a.meaning(s).s >>
    [] SKIP OD

THREAD LosePending() =                          % Drop a pending failed a
    DO true => << VAR a :IN pending | pending := pending - {a} >> OD

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. The body of `Do` is a single atomic action which happens between the invocation and the return; if `down` is true during that interval, one possible outcome of the body is to raise `failed`.

```
MODULE AMO2 [ as in AMO ] EXPORT Do =

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

PROC Do (a) -> V RAISES {failed} = <<
% Either do the action or raise failed.
% Raise failed only if the system is down sometime during the execution.
        VAR vs := a.meaning(s) | s := vs.s; RET vs.v
    []  down => pending := pending + {a}; RAISE failed >>

% Threads DoPending and LosePending as in AMO

THREAD Fail() = DO << down := true >>; << down := false >> OD
% Happens whenever a node crashes or the network fails.

END AMO2
```

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.

Primary Copy

The primary copy algorithm we sketch here is based on one invented by Liskov and Oki.[1] It implements a replicated state machine along the lines described in handout 26, and uses the Paxos consensus algorithm to decide the sequence of state machine actions. 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 `j` 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.

[1] B. Liskov and B. Oki, Viewstamped replication: A new primary copy method to support highly available distributed systems, *Proc. 7th ACM Conference on Principles of Distributed Computing*, Aug. 1988.

```
MODULE PrimaryCopy [                                    % implements AMO
      as in AMO
      C,                                                % Client names
      R ] EXPORT Do =                                   % Replica (server) names


TYPE K          =  ENUM[req, resp]                      % Kind of message
     X          =  ENUM[failed]                         % eXception result
     D          =  (A + V + X)                          % Data in message
     ID         =  Int                                  % Ids for messages
     P          =  (R + C)                              % All process names
     M          =  [sp: P, rp: P,                       % Message: sender. receiver,
                    id, k, d]
     Phase      =  ENUM[idle, trying]
     AID        =  [a, id]                              % Action plus UID
     J          =  Int                                  % Action index: 1, 2, ...
     VS         =  [v, s]                               % result of meaning
```

There is a separate instance of consensus for each action index `J`. Its outcome records the agreed-upon action and its index. By a slight abuse of Spec, we describe this with the `Actions` function from `J` to instances of the `Consensus` module. Consensus is run by the processes in `R`. In a real system the primary would also be both the leader and an agent of the consensus algorithm, and its state would normally include the outcomes of all the already decided actions as well as the next available action index.

We abstract the communication as a set of messages in transit among all the clients and replicas. This could be implemented by a set of the unreliable channels of handout 20, one in each direction for each client-server pair; this is the way most real systems do it. Note that the channel can lose both requests and responses. The channel connects the `Do` procedure with the server. The `Do` procedure, which is the client side of the channel, deals with these losses by assigning each action an `ID` that identifies it unambiguously, so that the server will only execute the action once. If there's a failure, the result value may still be lost; in this case `Do` raises `failed` as required by the `AMO` spec.

```
VAR  Actions    :  J -> Consensus[AID]
     msgs       :  SET M  := {}                         % messages in transit
     used       :  SET ID := {}                         % ID's used so far

PROC Do(a, c) -> V RAISES {failed} =
     IF   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);
          IF   VAR d := Receive(primary, c, id, resp) |
               IF d IS V => RET d [*] RAISE failed FI
          []   RAISE failed                             % if timeout on response
          FI
     FI

THREAD Actions(r) =                                     % one for each server
  VAR                                                   % state of primary
     c, id, a                                           % of current request
     j,                                                 % action index to try
     phase |
  DO % Do some enabled action
     << phase = idle => VAR c', id', d |                % Primary: receive request
        d := Receive(c', r, id', req); d IS A =>
             c := c'; id := id'; a := d;                % record request
             j := 0; phase := trying >>
  [] << phase = trying /\ j = 0 =>                      % Primary: choose a j
        VAR j' |   j' > 0 /\ Actions(j').Outcome() = nil
                /\ Actions(j'-1).Outcome() # nil =>
             IF   id IN {j1 :IN 0 .. j'-1 | |           % has id been done already?
                         Actions(j1).Outcome().id} => % if so, return failed
                Send(r, c, id, resp, failed); phase := idle >>
          [*]                                           % if not, start consensus
             j := j'; actions(j).Allow(AID{a, id})
     >>
  [] << VAR aid |   phase = trying /\ j # 0            % wait for consensus
                 /\ Actions(j).Outcome() = aid =>       %   to reach an outcome
        IF   aid.id = id =>                             % consensus: action j = a
             phase := idle; Send(r, c, id, resp, Value(j))
        [*]  j := 0                                     % another action got j
        FI
     >>
  [] << phase := idle >>                                % Crash
  OD

FUNC Value(j) -> V =
% Compute value returned by j'th action; needs all outcomes <= j
     RET Apply({j' :IN 1 .. j | | Actions(j').Outcome().a}, S.s0()).v

FUNC Apply(aq: SEQ A, s) -> VS =
% What these actions do, starting at s. Compose them all, then apply the result
     VAR allA := (* : (\ vs | aq.meaning(vs.s)}) | RET allA(VS{nil, s})

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

END PrimaryCopy
```

This version of the implementation doesn't maintain the current state at all, but reconstructs it explicitly from the sequence of actions. In a real system, the primary maintains both its idea of the last action index `j`, and an up-to-date state `s` that satisfies the invariant:

```
INVARIANT s = Apply({j' :IN 1 .. j | | Actions(j').Outcome().a}, S.s0()).s
```

This means that once the primary has obtained consensus on the action for the next `j`, it can update its state and return the corresponding result. If it doesn't obtain this consensus, then it isn't a legitimate primary. It needs to find out whether it should still be primary, and if so, bring its state up to date. The `CatchUp` procedure does this; we omit its code.

```
THREAD Actions(r) =                                     % one for each server
  VAR                                                   % state of primary
     c, id, a                                           % of current request
     j,                                                 % current action index
     s,                                                 % state from actions thru j
     phase |
  DO % Do some enabled action
     << phase = idle => VAR c', id', d |                % Primary: receive request
        d := Receive(c', r, id', req); d IS A =>
             c := c'; id := id'; a := d;                % record request
             IF   id IN {j1 :IN 0 .. j-1 | |            % has id been done already?
                         Actions(j1).Outcome().id} => % if so, return failed
                Send(r, c, id, resp, failed); phase := idle >>
          [*]                                           % if not, start consensus
             phase := trying; actions(j+1).Allow(AID{a, id}) >>
     >>
  [] << VAR aid |   phase = trying                     % wait for consensus
                 /\ Actions(j+1).Outcome() = aid =>     %   to reach an outcome
        IF   aid.id = id =>                             % consensus: action j = a
             j := j + 1; vs := a.meaning(s); s := vs.s;
             phase := idle; Send(r, c, id, resp, vs.v)
        [*]  CatchUp()                                  % another action got j
        FI
     >>
  [] << phase := idle >>                                % Crash
  OD
```

Voting

The voting algorithm sketched here is based on one invented by Dave Gifford.[2] The idea is that each replica has some version of the state. Versions are indexed by `J` just as in `PrimaryCopy` and each `Do` produces a new version. To read, you read the state of some copy of the latest version. To write, you find a copy of the current (latest) version, apply the action to create a new version, and write the new version into enough replicas. A real system does the updates in place, applying the action to enough replicas of the current version; it may have to bring some replicas up to date first.

The definition of 'enough' must ensure that both reads and writes find the latest version. The standard way to do this is to insist that both examine a majority of the replicas, where 'majority' is defined so that any two majorities intersect. Here majority is renamed 'quorum' to emphasize the fact that it may not be a numerical majority, and we allow for separate read and write quorums, since we only need to assure that any read or write sees any previous write, not necessarily any previous read. This distinction allows us to bias the code to make reads easier at the expense of writes, or vice versa. For example, we could make every replica a read quorum; then the only write quorum is all the replicas. This choice makes it easy to do a read, since you only need to reach one replica. On the other hand, writes are expensive, and in fact impossible if even one replica is down.

We abstract away from the details of communication and atomicity. As it was originally proposed by Gifford, and as it is presented here, the algorithm assumes that all the replicas can be updated atomically by a write, and that a replica can be read atomically. These atomic operations can be implemented by the distributed transactions of handout 27.

```
MODULE Voting [ as in AMO, R ] EXPORT Do =          % Replica (server) names

TYPE QS          =  SET SET R                       % Quorum Sets
     J           =  Int                             % Version number: 1, 2, ...

VAR  state       :  R -> S := (* -> S.s0())         % States of replicas
     vn          :  R -> J := (* -> 0)              % Version Numbers of replicas
     rqs         :  QS := {}                        % Read QuorumS
     wqs         :  QS := {}                        % Write QuorumS
     % rqs and wqs are initialized by the Init thread

APROC Do(a) -> V = <<
     IF   ReadOnly(a) =>                            % Read, not update
          VAR rq :IN rqs,
              j := {r' :IN rq | | vn(r')}.max, r | vn(r) = j =>
               RET a.meaning(state(r)).v
     [*]  VAR wq :IN wqs,                           % Update action
              j := {r' :IN wq | | vn(r')}.max, r | vn(r) = j =>
               j := j + 1;                           % new version number
               VAR vs := a.meaning(state(r)), s := vs.s |
                    DO VAR r' :IN wq | vn(r') < j =>state(r') := s; vn(r') := j OD;
                    RET vs.v
     FI >>

THREAD Init() = (rqs, wqs) := Quorums()

FUNC ReadOnly(a) -> Bool = RET (ALL s | a.meaning(s) = s)

APROC Quorums () -> (QS, QS) = <<
% Chooses sets of read and write quorums such that write quorum intersects
% every read or write quorum.
     VAR rqs': QS, wqs': QS | (ALL wq :IN wqs', q :IN rqs' + wqs' | q*wq # {}) =>
          RET (rqs', wqs') >>

END Voting
```

[2] D. Gifford, Weighted voting for replicated data. *ACM Operating Systems Review* **13**, 5 (Oct. 1979), pp 150-162.

Loosely consistent replication

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. We sketched a spec for this in the section on coherence in handout 12, and we repeat it here in a form that parallels our other specs.

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 below by allowing an action to be processed using any subsequence of the earlier actions to determine the response to the action. Such behavior is possible (though unlikely) in distributed naming systems such as Grapevine[3] or the domain name service[4]. The spec limits the nondeterminism by requiring an action's response to include the effects of all actions executed before the most recent `Sync`. If `Sync`'s are done reasonably frequently, the incoherence won't get out of hand. A paper by Lampson[5] goes into much more detail.

We write the spec in two ways. The first is in the style of handouts 7 and 12; it keeps track of all the possible states that the service can get into.

```
MODULE LooseReplication [as in AMO ] EXPORT Do, Sync =

VAR  s           :  S     := S.s0()                 % latest state
     ss          :  SET S := { S.s0()}              % all States since Sync

APROC Do(a) -> V = << VAR s0 :IN ss |               % choose a state for result
     s := a.meaning(s).s;
     ss := ss + {s' :IN ss | | a.meaning(s').s}
     RET a.meaning(s0).v >>

PROC Sync() = ss := {s}

END LooseReplication
```

The second keeps track explicitly of the actions done since the last `Sync`. `Apply` is defined in `PrimaryCopy`.

```
MODULE LooseReplication1 [as in AMO ] EXPORT Do, Sync =

VAR  s           :  S     := S.s0()                 % Sync'ed state
     aq          :  SEQ A := {}                     % all Actions since Sync

APROC Do(a) -> V = << VAR aq0 : SEQ A | aq0 <<= aq | % choose actions for result
     aq := aq + {a}; RET Apply(aq0 + {a}, s).v >>

PROC Sync() = s := Apply(aq, s).s; aq := {}

END LooseReplication1
```

Both of these specs are complicated by a perhaps ill-advised insistence that the result of `Do` should include the argument action.

[3] A. Birrell at al., Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4 (Apr. 1982), pp 260-274.
[4] RFC 1034/5.
[5] B. Lampson, Designing a global name service, *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, 1986, pp 1-10.