28. Availability and Replication

This handout explains the basic issues in building highly available computer systems, and describes in some detail the specs and code for a replicated service with state.

What is availability?

A system is available if it delivers service promptly. Exactly what this means is something that has to be specified. For example, the spec might say that an ATM must deliver money from a local bank account to the user within 15 seconds, or that an airline reservation system must respond to user input within 1 second. The definition of availability is the fraction of offered load that gets prompt service; usually it's more convenient to measure the probability *p* that a request is not serviced promptly.

If requests come in at a certain rate, say 1/minute, with a memoryless distribution (that is, what happens to one request doesn't depend on other requests; a tossed coin is memoryless, for example), then *p* is also the probability that not all requests arriving in one minute get service. If this probability is small then the time between bad minutes is 1/*p* minutes. This is called the 'mean time to failure' or MTTF; sometimes 'mean time between failures' or MTBF is used instead. Changing the time scale of course doesn't change the MTTF: the probability of a bad hour is 60*p*, so the time between bad hours is 1/60*p* hours = 1/*p* minutes. If *p* = .00001 then there are 5 bad minutes per year. Usually this is described as 99.999% availability, or '5-nines' availability.

The definition of 'available' is important. In a big system, for example, something is always broken, and usually we care about the service that one stream of customers sees rather than about whether the system is perfect, so we use the availability of one terminal to measure the MTTF. If you are writing or signing a contract, be sure that you understand the definition.

We focus on systems that fail and are repaired. In the simplest model, the system provides no service while it is failed. After it's repaired, it provides perfect service until it fails again. If MTTF is the mean time to failure and MTTR is the mean time to repair, then the availability is

$$p = \text{MTTR}/(\text{MTTF} + \text{MTTR})$$

If MTTR/MTTF is small, we have approximately

$$p = \text{MTTR}/\text{MTTF}$$

Thus the important parameter is the ratio of repair time to uptime. Note that doubling MTTF halves *p*, and so does halving the MTTR. The two factors are equally important. This simple point is often overlooked.

Redundancy

There are basically two ways to make a system available. One is to build it out of components that fail very seldom. This is good if you can do it, because it keeps the system simple. However, if there are *n* components and each fails independently with small probability p_c, then the system fails with probability $n\,p_c$. As *n* grows, this number grows too. Furthermore, it is often expensive to make highly reliable components.

The other way to make a system available is to use redundancy, so that the system can work even if some of its components have failed. There are two main patterns of redundancy: retry and replication.

Retry is redundancy in time: fail, repair, and try again. If failures are intermittent, repair doesn't require any action. In this case 1/MTBF is the probability of failure, and MTTR is the time required to detect the failure and try again. Often the failure detector is a timeout; then the MTTR is the timeout interval plus the retry time. Thus in retry, timeouts are critical to availability.

Replication is physical redundancy, or redundancy in space: have several copies, so that one can do the work even if another fails. The most common form of replication is 'primary-backup' or 'hot standby', in which the system normally uses the primary component, but 'fails over' to a backup if the primary fails. This is very much like retry: the MTTR is the failover time, which is the time to detect the failure plus the time to make the backup live. This is a completely general form of redundancy. Error correcting codes are a more specialized form. Two familiar examples are the Hamming codes used in RAM and the parity used in RAID disks.

These examples illustrate the application-dependent nature of specialized replication. A Hamming code needs log *n* check bits to protect *n* – log *n* data bits. A RAID code needs 1 check bit to protect any number of data bits. Why the difference? The RAID code is an 'erasure code'; it assumes that a data bit can have one of three values: 0, 1, and `error`. Parity means that the `xor` of all the bits is 0, so that any bit is equal to the `xor` of all the other bits. Thus any single `error` bit can be reconstructed from the others. This scheme is appropriate for disks, where there's already a very strong code detecting errors in a single sector. A Hamming code, on the other hand, needs many more check bits to detect which bit is bad as well as its correct value.

Another completely general form of replication is to have several replicas that operate in lockstep and interact with the rest of the world only between steps. At the end of each step, compare the outputs of the replicas. If there's a majority for some output value, that value is the output of the replicated system, and any replica that produced a different value is declared faulty and should be repaired. At least three replicas are needed for this to work; when there are exactly three it's called 'triple modular redundancy', TMR for short. A common variation that simplifies the handling of outputs is 'pair and spare', which uses four replicas arranged in two pairs. If the outputs of a pair disagree, it is declared faulty and the other pair's output is the system output.

A computer system has three major components: processing, storage, and communication. Here is how to apply redundancy to each of them.

- In communication, intermittent errors are common and retry is simply retransmitting a message. If messages can take different paths, even the total failure of a component often looks like an intermittent error because a retry will use different components. It's also possible to use error-correcting codes (called 'forward error correction' in this context), but usually the error rate is low enough that this isn't cost effective.
- In storage, retry is not so easy but error correcting codes still work well. ECC memory using Hamming codes, the elaborate codes used on disk drives, and RAID disks are all examples of this. Straightforward replication, usually called 'mirroring', is also popular.
- In processing, error correcting codes usually can't handle arbitrary state transitions. Retry is only possible if you have the old state, so it's usually coded in a transaction system. The replicated state machines that we studied in handout 18 are fully general, however, and can make any kind of processing highly available. Using these methods to replicate a processor at

the instruction set level is tricky but possible.[1] People also use lockstep replication at the instruction level, usually pair-and-spare, but such systems can't use standard components above the chip level, and it's very expensive to engineer them without single points of failure. As a result, they are expensive and not very successful.

War stories

Availability is a property of an entire system, hardware, software, and operations. There are lots of ways that things can go wrong. It's instructive to study some examples.

Ariane crash

The first flight of the European Space Agency's Ariane 5 rocket self-destructed 40 seconds into the flight. The sequence of events that led to this $400 million failure is instructive. In reverse temporal order, it is roughly as follows, as described in the report of the board of inquiry.[2]

1. The vehicle self-destructed because the solid fuel boosters started to separate from the main vehicle. This decision to self-destruct was part of the design and was carried out correctly.
2. The boosters separated because of high aerodynamic loads resulting from an angle of attack of more than 20 degrees.
3. This angle of attack was caused by full nozzle deflections of the solid boosters and the main engine.
4. The nozzle deflections were commanded by the on board computer (OBC) software on the basis of data transmitted by the active inertial reference system (SRI 2; the abbreviation is from the French for 'inertial reference system'). Part of the data for that time did not consist of proper flight data, but rather showed a diagnostic bit pattern of the computer of SRI 2, which was interpreted by the OBC as flight data.
5. SRI 2 did not send correct flight data because the unit had declared a failure due to a software exception.
6. The OBC could not switch to the back-up SRI (SRI 1) because that unit had already ceased to function during the previous data cycle (72-millisecond period) for the same reason as SRI 2.
7. Both units shut down because of uncaught internal software exceptions. In the event of any kind of exception, according to the system spec, the failure should be indicated on the data bus, the failure context should be stored in an EEPROM memory (which was recovered and read out), and, finally, the SRI processor should be shut down. This duly happened.
8. The internal SRI software exception was caused during execution of a data conversion from a 64-bit floating-point number to a 16-bit signed integer value. The value of the floating-point number was greater than what could be represented by a 16-bit signed integer. The result was an operand error. The data conversion instructions (in Ada code) were not protected from causing operand errors, although other conversions of comparable variables in the same place in the code were protected. It was a deliberate design decision not to protect this conversion, made because the protection is not free, and analysis had shown that overflow was impossible. In retrospect, of course, we know that the analysis was faulty; since it was not preserved, we don't know what was wrong with it.
9. The error occurred in a part of the software that controls only the alignment of the strap-down inertial platform. The results computed by this software module are meaningful only before liftoff. After liftoff, this function serves no purpose. The alignment function is operative for 50 seconds after initiation of the flight mode of the SRIs. This initiation happens 3 seconds before liftoff for Ariane 5. Consequently, when liftoff occurs, the function continues for approximately 40 seconds of flight. This time sequence is based on a requirement of Ariane 4 that is not shared by Ariane 5. It was left in to minimize changes to the well-tested Ariane 4 software, on the grounds that changes are likely to introduce bugs.
10. The operand error occurred because of an unexpected high value of an internal alignment function result, called BH (horizontal bias), which is related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time. The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values. There is no evidence that any trajectory data were used to analyze the behavior of the unprotected variables, and it is even more important to note that it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specifications.

It was the decision to shut down the processor that finally proved fatal. Restart is not feasible since attitude is too difficult to recalculate after a processor shutdown; therefore, the SRI becomes useless. The reason behind this drastic action lies in the custom within the Ariane program of addressing only random hardware failures. From this point of view, exception- or error-handling mechanisms are designed for random hardware failures, which can quite rationally be handled by a backup system. But a deterministic bug in software will happen in the backup system as well.

Maxc/Alto memory

The following extended saga of fault tolerance in computer RAM happened to my colleagues in the Computer Systems Laboratory of the Xerox Palo Alto Research Center. Many other people have had some of these experiences.

One of the lab's first projects (in 1971) was to build a time-sharing computer system named Maxc. Intel had just started to sell a 1024-bit semiconductor RAM chip[3], the Intel 1103, and it promised to be a cheap and reliable way to build the main memory. Of course, since it was new, we didn't know whether it would really work. However, we knew that for about 20% overhead we could use Hamming codes to implement single error correction and double error detection, so that the memory system would work even if individual chips hard a rather high failure rate. We did this, and the memory was solid as a rock. We never saw any failures, or even any double errors.

When the time came to design the Alto personal workstation in 1972, we used the same 1103 chips, and indeed the same memory boards. However, the Alto memory was much smaller (128

[1] Hypervisor-based fault tolerance, T. Bressoud and F. Schneider; *ACM Transactions on. Computing Systems* **14**, 1 (Feb. 1996), pp 80 – 107.
[2] This report is a model of clarity and conciseness. You can find it at http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html and a summary at http://www.siam.org/siamnews/general/ariane.htm.
[3] One million times smaller than the tate-of-the-art RAM chip of 2002.

KB instead of 3 MB) and had 16 bit words rather than the 40 bit words of Maxc. As a result, error correction would have added much more overhead, so we left it out; we did provide a parity bit for each word. For about 6 months the machines performed flawlessly, running a fairly vanilla minicomputer operating system that we had built, which provided a terminal on the screen that emulated a teletype.

It was only when we started to run the Bravo full-screen editor (the prototype for Microsoft Word) that we started to get parity errors. These errors were puzzling, because the chips were identical to those used without incident in Maxc. When we looked closely at the Maxc system, however, we discovered that although the ECC circuits had been designed to report both corrected errors and uncorrectable errors, the software logged only uncorrectable errors; corrected errors were being ignored. When logging of corrected errors was implemented, it turned out that the 1024-bit chips were actually failing quite often, and the error-correction circuitry was working hard to set things right.[4]

Investigation revealed that 1103's are pattern-sensitive: sometimes a bit will flip when the values of surrounding bits are just so. The reason we didn't see them on the Alto in the first 6 months is that you just don't get enough patterns on a single-user machine that isn't being very heavily used. Bravo put up lots of interesting stuff on the screen, which used about half the main memory to store values for its pixels, and thus Bravo made enough different patterns to tickle the chips. With some effort, we were able to write memory test programs that ran on the Alto, using lots of random test patterns, and also found errors. We never saw these errors in the routine testing that we did when the boards were manufactured.

Lesson: Fault-tolerant systems tend to become fault-intolerant, because faults that are tolerated don't get fixed. It's essential to monitor the faults and repair the faulty components even though the system is still working perfectly. Without monitoring, there's no way to know whether the system is operating with a large or a small safety margin.

When we built the Alto 2 two years later in 1975, we used 4k RAM chips, and because of the painful experience with the 1103, we did put in error correction. The machine worked flawlessly. Two years later, however, we discovered that in one-quarter of the memory, neither error correction nor parity was working at all. The chips were much better that 1103's, and in addition, many single-bit errors don't actually cause any observed failure of the software. On Alto 1 we knew about every single-bit error because of the parity. On Alto 2 in 1/4 of the memory we didn't know. Perhaps there were some failures that had no visible impact. Perhaps there were failures that crashed programs, but they were attributed to bugs in the software.

Lesson: To test a fault-tolerant system, you have to inject all the faults the system is supposed to tolerate. You also need to detect all faults, and you have to test the detection mechanism as well.

I believe this is why most PC manufacturers don't put parity on the memory: it isn't really needed because chips are pretty reliable, and if parity errors are reported the PC manufacturer gets blamed, whereas if random things happen Microsoft gets blamed.

[4] A couple of years later we had a similar problem with Maxc. In early January people noticed that the machine seemed to be slow. After a while, someone looked at the console log and discovered that over the holidays the memory had developed a permanent double (uncorrectable) error. The software found this error and reconfigured the memory without the bad region; this excluded one quarter of the memory from the running system, which considerably increased the amount of paging. Normally no one looked at the console log, so no one knew that this had happened.

Lesson: Beauty is in the eye of the beholder. The various parties involved in the decisions about how much failure detection and recovery to code do not always have the same interests.

Replication

In the remainder of this handout we present specs and code for a variety of replication techniques. We start with two specs of a "strongly consistent" replicated service, which looks almost like a single copy to its clients. The complication is that some client requests can fail; the second spec constrains the failure behavior more than the first. Then we give two codes, one based on primary copy and the other based on voting. Finally, we give a spec of a "loosely consistent" service, which is much weaker but allows much cheaper highly available code.

Specs for consistent replication

A 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 (for example, if the action leaves an unambiguous record in the state, such as a sequence number), but the `failed` response gives no information. The idea is that a `failed` response may be caused by failure of the replica doing the action, or of the communication channel between the client and the service.

The first spec 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 spec still allows actions with `failed` responses to happen at any later time. However, it allows a `failed` response only if the system fails (or is recovering from a failure) during the execution of an action.

In practice, some constraints on when failed actions are performed would be desirable, but it seems hard to write a general spec of such constraints that applies to a wide range of code. 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 spec, which allows `failed` responses at any time. It is modeled on the spec for sequential transactions in handouts 7 and 19.

```
MODULE Replication [
    V,                                          % Value
    S WITH { s0: () -> S }                      % State
    ] EXPORT Do =

TYPE VS          =  [v, s]
    A            =  S -> VS                     % Action

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(s) | s := vs.s; RET vs.v
    []   pending \/ := {a}; RAISE failed >>
```

```
THREAD DoPending() =                                % Do or drop a pending failed a
    DO << VAR a :IN pending |
        pending - := {a};
        BEGIN s := a(s).s [] SKIP END >>            % Do a or drop it
    [] SKIP OD

END Replication
```

Here is the second spec. Intuitively, we would like a failed response only if the service fails (by 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`. In the spec above, `Do` is an `APROC`; that is, there is a single atomic action in the module's interface. In the second spec below, `Do` is not an `APROC` but a `PROC`; that is, there are two atomic actions in the interface, one to invoke `Do` and one for its return.

Note that an action that has made it into `pending` can be executed at an arbitrary later time, perhaps when `down = false`.

`MODULE Replication2` `[ V, S` as in `Replication ] EXPORT Do =`

```
TYPE VS          =  [v, s]
    A            =  S -> VS                     % Action

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} = <<            % Do a or raise failed
% Raise failed only if the system is down sometime during the execution. Note that this isn't an APROC
        VAR vs := a(s) | s := vs.s; RET vs.v
    []  down => pending \/ := {a}; RAISE failed >>
```

`%` Thread `DoPending` as in `Replication`

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

```
END Replication2
```

There are two general ways of coding 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 give here is based on one invented by Liskov and Oki.[5] It codes a replicated state machine along the lines described in handout 18, using the Paxos consensus algorithm to decide the sequence of state machine actions. When things are working well, the clients send action 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

[5] 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.

responds to the client. We only assign an index `j` to an action if all prior indices have been assigned to actions, and no later ones.

For simplicity, we assume that every action is unique, and use the action to identify all the messages and outcomes associated with it. In practice, clients accomplish this by tagging each action with a unique ID and use the ID for this purpose.

`MODULE PrimaryCopy` `[` % implements `Replication`

```
    V, S as in Replication
    C,                                          % Client names
    R ] EXPORT Do =                             % Replica (server) names

TYPE VS          =  [v, s]
    A            =  S -> VS                     % Action
    X            =  ENUM[failed]                % eXception result
    Data         =  (Null + V + X)              % Data in message
    P            =  (R + C)                     % All process names
    M            =  [sp: P, rp: P, a, data]     % Message: sender, rcvr, action, data
    J            =  Nat                         % Action index: 1, 2, ...
```

There is a separate instance of consensus for each action index `J`. Its outcome records the agreed-upon `j`th action. We achieve this by making the `Consensus` module of handout 18 into a `CLASS` with `A` as `V`. The `Actions` function maps from `J` to instances of the class. The processes in `R` run consensus. 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 (or at least the recent ones) as well as the next available action index. This means that all the old outcomes will be available, so that `Outcome()` will never return `nil` for one of them. We assume this in what follows, and accordingly make `outcome` a function.

`CLASS ReplCons` `EXPORT allow, outcome =`

```
VAR outcom   : (A + Null) := nil

APROC allow(a) = << outcome = nil => outcom := a [] SKIP >>
FUNC  outcome() -> (A + Null) = << RET outcom >>

END ReplCons
```

We abstract the communication as a set of messages in transit among all the clients and replicas. This could be coded by a set of the unreliable channels of handout 21, one in each direction for each client-replica pair; this is the way most real systems do it. Note that the channel can lose or duplicate both requests and responses. The channel connects the `Do` procedure with the replica. The `Do` procedure, which is the client side of the channel, deals with losses by retransmitting. If there's a failure, the result value may be lost; in this case `Do` raises `failed` as required by the `Replication` spec.

The client code keeps trying to get a replica to handle its request. The replica proceeds as though it is the primary. If there's more than one primary, there will be contention for action indexes, so this is not desirable. Since we are using Paxos, there should be only one primary at a time. In fact, the primary and the Paxos leader should be the same. Usually the primary has a lease, which has some advantages discussed later. For simplicity, we show each replica handling only one request at a time; in practice, of course, they could be batched. In spite of this, there can be lots of requests in progress at a time, since several replicas may be handling client request simultaneously if there is confusion about who is the primary.

We begin with code in which the replicas only keep track of the actions, that is, the results of consensus. This is not very practical, since it means that they have to recompute the current state from scratch for every request, but it is simple; we did the same thing when introducing transactions in handout 7. Later we consider the complications of keeping track of the current state.

```
VAR  actions     :  J -> ReplCons := InitActions()
     msgs        :  SEQ M  := {}                        % multiset of messages in transit
     working     :  P -> (A + Null) := {}               % history, for abstraction function

% ABSTRACTION FUNCTION:
     Replication.s = AllActions(LastJ())(S.s0()).s
     Replication.pending =   working.rng \/ {m :IN msgs | m.data = nil || m.a}
                           – Outcome.rng – {nil}

% INVARIANT: (ALL j :IN 1 .. LastJ() | Outcome(j) # nil)

% The client
PROC Do(a, c) -> V RAISES {failed} =                    % First choose a new uid
     working(c) := a;                                   % Just for the abstraction function
     DO VAR primary: R |                                % Guess the current primary
          Send(c, primary, a, nil);
          VAR a', data | (primary, a', data) := Receive(c);
               IF   a' = a => IF data IS V => RET data [*] RAISE failed FI
               [*]  SKIP FI                             % Discard responses that aren't to a
     []   SKIP                                          % if timeout on response
     []   RAISE failed                                  % if too many retries
     OD; working(c) := nil                              % Just for the abstraction function

% The server replicas
THREAD DoActions(r) =                                   % one for each replica
     DO VAR c, a, data |                                % of current request
          << (c,a,data):=Receive(r); working(r):=a >>;  % Primary: receive request
          data := DoAction(id, a); Send(r, c, a, data)  % Do it and send response
          working(r) := nil                             % Just for the abstraction function
     OD

PROC DoAction(id, a) -> Data =
  DO VAR j |                                            % Keep trying until id is done.
     j := LastJ();                                      % Find last completed j
     IF a IN Outcome.rng => RET failed                  % Has a been done already? If so, failed
     [*]  j + := 1; actions(j).allow(a);                % No. Try for consensus on a=action j
          Outcome(j) # nil =>                           % Wait for consensus
               IF   Outcome(j) = a => RET Value(j)      % If we got j, Return its result.
               [*]  SKIP FI                             % Another action got j. Try again.
     FI
  OD


% These routines compute useful functions of the action history.

FUNC Value(j) -> V = RET AllActions(j)(S.s0()).v
% Compute value returned by j'th action; needs all outcomes <= j

FUNC AllActions(j) -> A = RET Compose({j' :IN 1 .. j || Outcome(j')})
% The composition of all the actions through j. Type error if any of them is nil.

FUNC Compose(aq: SEQ A) -> A =
     aq # {} => RET aq.head * (* : {a :IN aq.tail || (\ vs | a(vs.s))})
```



```
FUNC LastJ() -> J = RET {j' | Outcome(j') # nil}.max [*] RET 0
% Last j for which consensus has been reached.

FUNC Outcome(j) -> (A + Null) = RET actions(j).outcome()

PROC InitActions() -> (J -> ReplCons) =                 % Make a ReplCons for each j
     VAR acts: J -> ReplCons := {}, rc: ReplCons |
          DO VAR j | ~ acts!j => acts(j) := rc.new OD; RET acts

% Here is the standard unreliable channel.
APROC Send(p1, p2, id, data) = << msgs := msgs \/ {M{p1, p2, id, data}} >>
APROC Receive(p) -> (P, ID, Data) = << VAR m :IN msgs |      % Receive msg for p
     m.rp = p => msgs - := {m}; RET (m.sp, m.id, m.data) >>
THREAD LoseOrDup() =
     DO << VAR m :IN msgs | BEGIN msgs - :={m} [] msgs \/ :={m} END >> [] SKIP OD

END PrimaryCopy
```

There is no explicit code for crashes. A crash simply resets the control state. For the client, this has the same practical effect as getting a failed response: you don't know whether the action happened or not. For the replica, either it got consensus or it didn't. If it did, the action has happened; if not, it hasn't. Either way, client will keep trying if the replica hasn't already sent a response that isn't lost in the channel. The client may see a failed response or it may get the result value.

Instead of failing if the action has already been done, we could try to return the proper result. It's unreasonably expensive to guarantee to always do this, but it's quite practical to do it for recent requests. This changes one line of `DoAction`:

```
     IF a IN Outcome.rng =>
          BEGIN RET failed [] RET Value({j | Outcome(j) = a}.choose) END
```

This code is completely non-deterministic about retransmissions. As usual, it's necessary to be prudent in practice, especially since talking to too many replicas may cause needless failed responses. We have omitted any details about how the client finds the current primary; in practice, if the client talks to a replica that isn't the primary, that replica can redirect the client to the current primary. Of course, this redirection might happen several times if the system is unstable.

In this code replicas keep actions forever, both so that they can reconstruct the state and so that they can detect duplicate requests. When replicas keep the current state they don't need all the actions for that, but they still need them to detect duplicates. The reliable messages of handout 26 can't help with this, because they work only when a sender is talking to a single receiver, and here there are many receivers, one for each replica. . Real systems usually don't keep actions forever. Instead, they time them out, and often they tie each action to the current choice of primary, so that the action gets a failed response if the primary changes during its execution. To reconstruct the state of a very old replica, they copy the entire state from some other replica and then apply the most recent actions to bring it fully up to date.

The code above doesn't keep track of either the current state or the current action, but reconstructs them explicitly from the sequence of actions, using `LastJ` and `AllActions`. In a real system, the primary maintains both its idea of the last action index `j` and a corresponding state `s`. These satisfy the obvious invariant. In addition, the primary's `j` is the latest one, except while the primary is getting consensus, which it can't do atomically:

```
VAR  jr          :  R -> J := {* -> 0}
     sr          :  R -> S := {* -> S.s0()}
```

```
INVARIANT (ALL r | sr(r) = AllActions(jr(r))(S.s0()).s)
INVARIANT jr(primary) = LastJ() \/ primary is getting consensus
```

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 the latter; we omit the code that chooses the primary. In practice we don't keep the entire action history, but catch up a severely outdated replica by copying the state from a current one; we omit this code as well.

```
PROC DoAction(id, a) -> Data =
  DO VAR j := jr(r) |                                  % Don't need to search for j.
    IF << a IN Outcome.rng => RET failed               % Has a been done already? If so, failed
    [*] j + := 1; actions(j).allow(a);                 % No. Try for consensus on a=action j
        Outcome(j) # nil =>                            % Wait for consensus
            IF Outcome(j)=a => VAR vs := a(sr(r)) |    % If we got j, return its result.
                << sr(r) := vs.s; jr(r) := j >>; RET vs.v
            [*] CatchUp(r) FI                          % Another action got j. Try again.
    FI
  OD
```

```
PROC Catchup(r) =                                      % Apply actions until you run out
    DO VAR j := jr(r) + 1, o := Outcome(j) |
        o = nil => RET;
        sr(r) := (o AS a)(sr(r)).s; jr(r) := j
    OD
```

Note that the primary is still running consensus for each action. This is necessary so that another replica can take over should the primary fail. It can, however, use the optimization for a sequence of consensus actions that is described in handout 18; this means that each consensus takes only one round-trip.

When they are running normally, the other replicas will run `Catchup` in the background, based on the information they get from the consensus. If a replica gets out of touch with the consensus, it can run the full `Catchup` to get back up to date.

We have assumed that a replica can do each action atomically. In general this will require the replica to use a transaction. The logging needed for the transaction can also provide the storage needed for the consensus outcomes.

A further optimization is for the primary to obtain a lease. As we saw in handout 18, this means that it can respond to read-only requests from its copy of the state, without needing to run consensus. Furthermore, the other replicas can be simple read-write memories rather than active agents; in particular, they can be disk drives. Of course, if the primary fails we have to find another computer to act as primary.

Voting

The voting algorithm sketched here is based on one invented by Dave Gifford.[6] 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,

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

and write the new version into enough replicas. A distributed transaction makes this operation atomic. 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.

Warning: Because `Voting` is built on distributed transactions, it isn't easy to compare it to `PrimaryCopy`, which is only built on the basic `Consensus` primitive.

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.

There are many other ways to arrange the quorums. One simple scheme is to arrange the processes in a rectangle, make each row a read quorum, and make each row-column pair a write quorum. For a square with n replicas, a read quorum has $n^{1/2}$ replicas and a write quorum $2\,n^{1/2} - 1$. By changing the shape of the rectangle you can favor reads or writes. If there are lots of replicas, these quorums are much smaller than a majority.

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**
13	**14**	**15**	**16**

Note that the failure of an entire quorum makes the system unavailable. So the price paid for small quorums is that a small number of failures makes the system fail.

We abstract away from the details of communication and atomicity. 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 coded by the distributed transactions of handout 27. The consensus that is necessary for replication is hiding in the two-phase commit.

The abstract state is the state of a current replica. The invariant says that every `rq` has a current version, there's a `wq` in which every version is current, and two replicas with the same version also have the same state.

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

TYPE QS          =  SET SET R                          % Quorum Sets
     RWQ         =  [r: QS, w: QS]
     J           =  Int                                % Version number: 1, 2, ...

VAR  sr          :  R -> S := (* -> S.s0())            % States of replicas
     jr          :  R -> J := (* -> 0)                 % Version Numbers of replicas
```

```
    rwq         := Quorums()                          % Read QuorumS

% ABSTRACTION FUNCTION: replication.s = sr({r | jr(r) = jr.rng.max}.choose)

% INVARIANT:    (ALL rq :IN rwq.r | jr.restrict(rq).rng.max = jr.rng.max)
             /\ (EXISTS wq :IN rwq.w | jr.restrict(wq).rng = (jr.rng.max}
             /\ (ALL r1, r2 | jr(r1) = jr(r2) ==> sr(r1) = sr(r2))

APROC Do(a) -> V = <<
    IF   ReadOnly(a) =>                               % Read, not update
         VAR rq :IN rwq.r,
              j := jr.restrict(rq).rng.max, r | jr(r) = j =>
              RET a(sr(r)).v
    []   VAR wq :IN rwq.w,                            % Update action
              j := jr.restrict(wq).rng.max, r | jr(r) = j =>
              j := j + 1;                             % new version number
              VAR vs := a(sr(r)), s := vs.s |
                   DO VAR r' :IN wq | jr(r') < j =>sr(r') := s; jr(r') := j OD;
                   RET vs.v
    FI >>

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

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

END Voting
```

Note that because the read and write quorums intersect, every read sees all the preceding writes. In addition, any two write quorums must intersect, to ensure that writes are done with increasing version numbers and that a write sees the state changes made by all preceding writes. When the quorums are simple majorities, every quorum is both a read and a write quorum, so this complication is taken care of automatically. In the square scheme, however, although a read quorum can be a single row, a write quorum *cannot* be a single column, even though that would intersect with every row. Instead, a write quorum must be a row plus a column.

It's possible to reconfigure the quorums during operation, provided that at least one of the new write quorums is made completely current.

```
APROC NewQuorums() = <<
    VAR new := Quorums(), j:= jr.rng.max, s:=sr({r | jr(r)=jr.rng.max}.choose) |
         VAR wq :IN new.w | DO VAR r :IN wq | jr(r) < j => sr(r) := s OD;
         rwq := new
```

Loosely consistent replication

Some services have availability and response time constraints that make it impossible to maintain sequential consistency, 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[7] like DNS, for distributing information such as password files, and for updating system binaries. We sketched a spec for this in the section on coherence in handout 12 on naming; we repeat it

[7] also called 'directories' in networks, and not to be confused with file system directories

here in a form that parallels our other specs. Another name for this kind of loose replication is 'eventual consistency'.

Propagating updates in the background means that when an action is processed, the replica processing it might not know about some earlier actions. `LooseRepl` reflects this by allowing *any* subsequence of the earlier actions to determine the response to an action. Such behavior is possible (though unlikely) in distributed naming systems such as Grapevine[8] or the domain name service[9]. The spec limits the nondeterminism by requiring a 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[10] goes into much more detail.

For this to make sense as the system evolves, the actions must be defined on every state, and the result must be independent of the order in which the actions are applied (that is, they must all commute). In addition, it's simpler if the actions are idempotent (for the same reason that idempotency simplifies transaction redo recovery), and we assume that as well. Thus

```
    (ALL aq: SEQ A, aa: SET A | aq.rng = aa ==> Compose(aq) = Compose(aa.seq))
```

You can always get idempotency by tagging each action with a unique ID, as we saw with transactions. To make the standard `read` and `write` operations on path names described in handout 12 commutative and idempotent, tag each name in the path name with a version number or timestamp, both in the actions and in the state.

We write the spec in two equivalent ways. The first is in the style of handout 7 on disks and file systems and handout 12 on naming; it keeps track of all the possible states that the service can get into. It defines `Sync` to pick *some* state later than the latest one at the start of the `Sync`. It would be simpler to define `Sync` as `ss := {s}` and get rid of `ssNew`, as we did in handout 7, but this is too strong for the code we have in mind. Furthermore, the extra strength doesn't help the clients. `DropFromSS` doesn't change the behavior of the spec, since it only drops states that might not be used anyway, but it does make it easier to write the abstraction function.

`MODULE LooseRepl` `[ V, S WITH {s0: ()->S] EXPORT Do, Sync =`

```
TYPE VS          =  [v, s]
     A           =  S -> VS                           % Action

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

APROC Do(a) -> V = <<
     s := a(s).s; ss := Extend(ss, a); ssNew := Extend(ssNew, a);
     VAR s0 :IN ss | RET a(s0).v >>                   % choose a state for result

PROC Sync() = ssNew := {s}; << VAR s0 :IN ssNew | ss := {s0} >>; ssNew := {}

THREAD DropFromSS() =
     DO << VAR s1 :IN ss, s2 :IN ssNew | ss - := {s1}; ssNew - := {s2} >>
     [] SKIP OD
```

[8] A. Birrell at al., Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4 (Apr. 1982), pp 260-274.
[9] RFC 1034/5. You can find these at http://www.rfc-editor.org/isi.html. If you search the database for them, you will see information about updates.
[10] B. Lampson, Designing a global name service, *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, 1986, pp 1-10. You can find this at http://research.microsoft.com/lampson.

```
FUNC Extend(ss: SET S, a) -> SET S = RET ss \/ {s' :IN ss || a(s').s}

END LooseRepl
```

The second spec is closer to the code. It remembers the state at the last Sync instead of the current state, and keeps track explicitly of the actions done since the last Sync. After a Sync all the actions that happened before the Sync started are included in s, together with some subset of later ones.

MODULE LooseRepl2 [V, SA WITH {s0: ()->SA] EXPORT Do, Sync =

```
TYPE S          = SA WITH {"+":=Apply}
     VS, A as in LooseRepl
VAR  s          :  S     := S.s0()          % synced State (not latest)
     aa         :  SET A := {}              % All Actions since last sync
     aaOld      :  SET A := {}              % All Actions between last two Syncs

APROC Do(a) -> V = <<
     VAR aa0 : SET A | aa0 <= aa \/ aaOld =>          % choose actions for result
          aa \/ := {a}; RET a((s + aa0)).v >>

PROC Sync() =
     << aaOld := aa; aa := {} >>; << s := s + aaOld; aaOld := {} >>

THREAD DropFromAA() =
     DO << VAR a :IN aa \/ aaOld | s := s + {a}; aa - := {a}; aaOld - := {aa} >>
     [] SKIP
     OD

FUNC Apply(s0, aa0: SET A) -> S = RET PrimaryCopy.Compose(aa0.seq)(s).s

END LooseRepl2
```

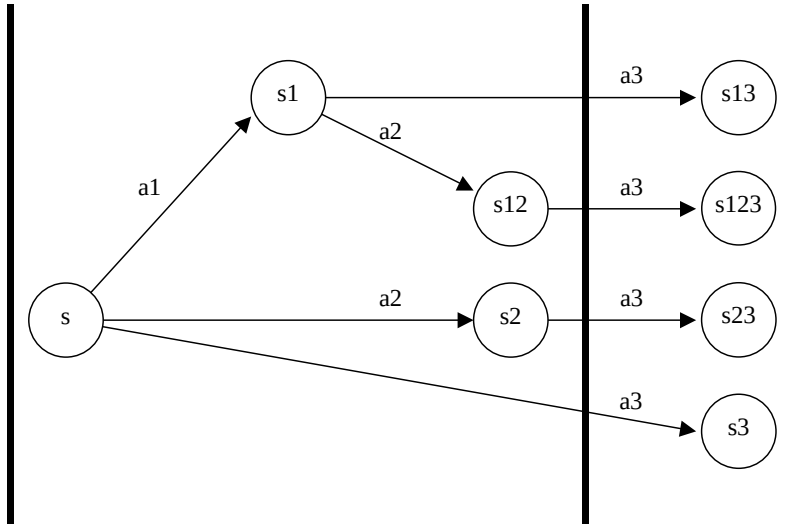
The picture shows how the set of possible states evolves as three actions are added. It assumes no actions are added while Sync 6 was running, so that the only state at the end of Sync 6 is s.

The abstraction function from LooseRepl2 to LooseRepl constructs the states from the Synced state and the actions:

```
ABSTRACTION FUNCTION
     LooseRepl.s     = s + aa
     LooseRepl.ss    = {aa1: SET A | aa1 <= aa || s + aa1}
     LooseRepl.ssNew = {aa1: SET A | aa1 <= aa || s + (aa1 \/ aaOld)}
```

We leave the abstraction function from LooseRepl to LooseRepl2 as an exercise.

The standard code has a set of replicas, each with a current state and a set of actions accumulated since the start of the last Sync; note that this is different from either spec. Typically actions have the form "set the value of name n to v". Any replica can execute a Do action. During normal operation the replicas send actions to each other with Gossip; more detailed code would send a (or a set of a's) from r1 to r2 in a message. Sync collects all the recent actions and distributes them to every replica. We omit the complications of catching up a replica that has missed some Syncs and of keeping track of the set of replicas.



Sync 6 Sync 7

MODULE LRImpl [as in Replication, % implements LooseRepl2
R] EXPORT Do, Sync = % Replica (server) names

```
TYPE VS          =  [v, s]
     A           =  S -> VS                  % Action
     J           =  NAT                      % Sync index: 1, 2, ...

VAR  jr          :  R -> J := {* -> 0}       % latest Sync here
     sr          :  R -> S := {* -> S.s0()}  % current State here
     hsrOld      :  R -> S := {* -> S.so()}  % history: state at last Sync
     hsOld       :  S := S.so()              % history: state at last Sync
     aar         :  R -> SET A := {* -> {}}  % actions since last Sync here

ABSTRACTION FUNCTION

APROC Do(a) -> V = << VAR r, vs := a(sr(r)) |
     aar(r) \/ := {a}; sr(r) := vs.s; RET vs.v >>

THREAD Gossip(r1, r2) =
     DO VAR a :IN aar(r1) - aar(r2) | aar(r2) \/ := a; sr(r2) := a(sr(r2))
     [] SKIP OD

PROC Sync() =
  VAR aa0: SET A := {},
      done: R -> Bool := {* -> false},
      j | j > jr.rng.max =>
     DO VAR r | jr(r) < j =>                          % first pass: collect all actions
          << jr(r) := j; aa0 \/ := aar(r); aar(r) := {} >> OD;
     DO VAR r | ~ done (r) =>                         % second pass: distribute all actions
          << sr(r) := sr(r) \/ aa0; done (r) := true >> OD


END LRImpl
```