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 MBTF is used instead. Changing the time scale of course doesn't change the MTTF: the probability of a bad hour is $60p$, so the time between bad hours is $1/60p$ hours = $1/p$ minutes. If $p = .00001$ then there are 5 bad minutes per year. In a big system 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.

We focus on systems that fail and are repaired. While the system is failed, it provides no service. 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

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

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.

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 weaker form of physical replication (that is, one that tolerates fewer failures) is an error correcting code. It's easy to apply this to storing and transmitting data; we cite some examples below.

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, component failures often look like intermittent errors 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). 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, 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.

[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

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 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 quite busy in setting things right.[3]

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.

[3] 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: 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 the software supplier gets blamed.

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, 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:

```
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`. 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.[4] 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 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, receiver,
    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 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

[4] 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.

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. Just as with 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 request 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. 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) := {}                 % Just 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 as 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 always guarantee to 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.

This version of the code 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:

```
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.

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

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 as 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.

Voting

The voting algorithm sketched here is based on one invented by Dave Gifford.[5] 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 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.

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

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

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)

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
```


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 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[6], 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 on naming, and we repeat it 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. 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[7] or the domain name service[8]. 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[9] 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.set = 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 would be simpler to define `Sync` as `ss := {s}` and get rid of `ssNew`, as we did in

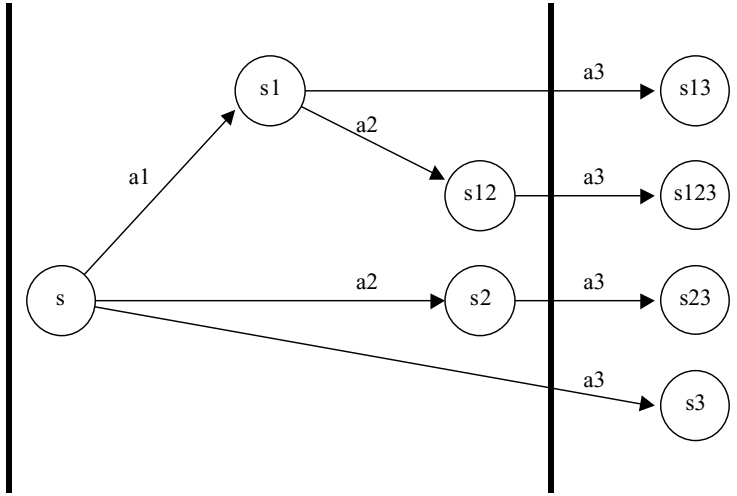
[6] also called 'directories' in networks, and not to be confused with file system directories
[7] A. Birrell at al., Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4 (Apr. 1982), pp 260-274.
[8] RFC 1034/5. You can find these at http://www.rfc-editor.org/isi.html. If you search the database for them, you will see information about updates.
[9] 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.

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

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

END LooseRepl
```

The second spec 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, assuming that no actions were 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.

```
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

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
```

30. Concurrent Caching

This handout presents several specs and codes for caches in concurrent systems. We begin with a spec for `CoherentMemory`, the kind of memory we would really like to have; it is just a function from addresses to data values. We also specify the `IncoherentMemory` that has fast code, but is not very nice to use. Then we show how to change `IncoherentMemory` so that it codes `CoherentMemory` with as little communication as possible. We describe various strategies, including invalidation-based and update-based strategies, and strategies using incoherent memory plus locking.

Since the various strategies used in practice have a lot in common, we unify the presentation using successive refinements. We start with cache code `GlobalImpl` that clearly works, but is not practical to code directly because it is extremely non-local. Then we refine `GlobalImpl` in stages to obtain (abstract versions of) practical code.

First we show how to use reader/writer locks to get a practical version of `GlobalImpl` called a coherent cache. We do this in two stages, an ideal cache `CurrentCaches` and a concrete cache `ExclusiveLocks`. The caches change the guards on internal actions of `IncoherentMemory` as well as on the external read and write actions, so they can't be coded externally, simply by adding a test before each read or write of `IncoherentMemory`, but require changes to its insides.

There is another way to use locks to get a different practical version of `GlobalImpl`, called `ExternalLocks`. The advantage of `ExternalLocks` is that the locking is decoupled from the internal actions of the memory system so that it can be coded separately, and hence `ExternalLocks` can run entirely in software on top of a memory system that only implements `IncoherentMemory`. In other words, `ExternalLocks` is a practical way to program coherent memory on a machine whose hardware provides only incoherent memory.

There are many practical codes for the methods that are described abstractly here. Most of them originated in the hardware of shared-memory multiprocessors.[1] It is also possible to code shared memory in software, relying on some combination of page faults from the virtual memory and checks supplied by the compiler. This is called 'distributed shared memory' or DSM.[2] Intermediate schemes do some of the work in hardware and some in software.[3] Many of the techniques have been re-invented for coherent distributed file systems.[4]

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1996, chapter 8, pp 635-754.
[2] K. Li and P. Hudak, Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* **7**, 4 (Nov. 1989), pp 321-359. For recent work in this active field see any ISCA, ASPLOS, OSDI, or SOSP proceedings.
[3] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: performance and cost. *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 314-324, April 1994 (http://www.cag.lcs.mit.edu/alewife/papers/soft-ext-isca94.html). Jeffrey Kuskin et al., The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994 (http://www-flash.stanford.edu/architecture/papers/ISCA94).
[4] M. Nelson et al., Caching in the Sprite network file system. *ACM Transactions on Computer Systems* **11**, 2 (Feb. 1993), pp 228-239. For recent work in this active field see any OSDI or SOSP proceedings.

All our code makes use of a global memory that is modeled as a function from addresses to data values; in other words, the spec for the global memory is simply `CoherentMemory`. This means that actual code may have a recursive structure, in which the top-level code for `CoherentMemory` using one of our algorithms contains a global memory that is coded with another algorithm and contains another global memory, etc. This recursion terminates only when we lose interest in another level of virtualization. For example,

a processor's memory may consist of a first level cache plus
a global memory made up of a second level cache plus
a global memory made up of a main memory plus
a global memory made up of a local swapping disk plus
a global memory made up of a file server ....

Specs

First we recall the spec for ordinary coherent memory. Then we give the spec for efficient but ugly incoherent memory. Finally, we discuss an alternative, less intuitive way of writing these specs.

Coherent memory

The first spec is for the memory that we really want, which ensures that all memory operations appear atomic. It is essentially the same as the `Memory` spec from Handout 5 on memory specs, except that `m` is defined to be total. In the literature, this is sometimes called a 'linearizable' memory.

```
MODULE CoherentMemory [P, A, V] EXPORT Read, Write =
% Arguments are Processors, Addresses and Data

TYPE M          =  A -> D SUCHTHAT (\ f: A->D | (ALL a | f!a))
VAR  m

APROC Read(p, a) -> D = << RET m(a) >>
APROC Write(p, a, d) = << m(a) := d >>

END CoherentMemory
```

From this point we drop the `a` argument and study a memory with just one location; that is, we study a cached register. Since everything about the specs and code holds independently for each address, we don't lose anything by doing this, and it reduces clutter. We also write the `p` argument as a subscript, again to make the specs easier to read. The previous spec becomes

```
MODULE CoherentMemory [P, V] EXPORT Read, Write =
% Arguments are Processors and Data

TYPE M          =  D                                          % Memory
VAR  m

APROC Read_p -> D = << RET m >>
APROC Write_p(d) = << m := d >>

END CoherentMemory
```

Of course, code usually has limits on the size of a cache, or other resource limitations that can only be expressed by considering all the addresses at once, but we will not study this kind of detail here.

Incoherent memory

The next spec describes the minimum guarantees made by hardware: there is a private cache for each processor, and internal actions that move data back and forth between caches and the main memory, and between different caches. The only guarantee is that data written to a cache is not overwritten in that cache by anyone else's data. However, there is no ordering on writes from the cache to main memory.

Since this is not enough to get any useful work done, we add a `Barrier` synchronization operation that forces the cache and memory to agree. This can be used after a `Write` to ensure that an update has been written back to main memory, and before a `Read` to ensure that the data being read is current. `Barrier` was called `Sync` when we studied disks and file systems in handout 7.

Note that `Read` has a guard `Live` that it makes no attempt to satisfy (hardware codes usually have an explicit flag called `valid`). Instead, there is another action `MtoC` that makes `Live` true. In a real system an attempt to do a `Read` will trigger a `MtoC` so that the `Read` can go ahead, but in Spec we can omit the direct linkage between the two actions and let the non-determinism do the work. We use this coding trick repeatedly in this handout. Another example is `Barrier`, which forces the cache to drop its data by waiting until `Drop` happens; if the cache is dirty, `Drop` will wait for `CtoM` to store its data into memory first.

You might think that this is just specsmanship and that a nondeterministic `MtoC` is silly, but in fact transferring data from `m` to `c` without a `Read` is called prefetching, and many codes do it under various conditions: because it's in the next block, or because a past reference sequence used it, or because the program executes a prefetch instruction. Saying that it can happen nondeterministically captures all of this behavior very simply.

We adopt the convention that an invalid cache entry has the value `nil`.

```
MODULE IncoherentMemory [P, A, V] EXPORT Read, Write, Barrier =

TYPE M          =  D                                  % Memory
     C          =  P -> (D + Null)                    % Cache

VAR  m          :  CoherentMemory.M                   % main memory
     c          := C{* -> nil}                        % local caches
     dirty      :  P -> Bool := {*->false}            % dirty flags

% INVARIANT Inv1: (ALL p | c!p)                       % each processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)        % dirty data is in the cache

APROC Read_p -> D = << Live_p => RET c_p >>           % MtoC gets data into cache
APROC Write_p(d)  = << c_p := d; dirty_p := true >>

APROC Barrier_p = << ~ Live_p => SKIP >>              % wait until not in cache

FUNC Live_p -> Bool = RET (c_p # nil)
```



```
% Internal actions

THREAD Internal_p = DO MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p [] SKIP OD

APROC MtoC_p = << ~ dirty_p => c_p := m >>                % copy memory to cache
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >> % copy cache to memory
APROC CtoC_p,p' = << ~ dirty_p' /\ Live_p => c_p' := c_p >> % copy from cache p to p'
APROC Drop_p = << ~ dirty_p => c_p := nil >>              % drop clean data from cache

END IncoherentMemory
```

In real code some of these actions may be combined. For example, if the cache is dirty, a real barrier operation may do `CtoM; Barrier; MtoC` by just storing the data. These combinations don't introduce any new behavior, however, and it's simplest to study the minimum set of actions presented here.

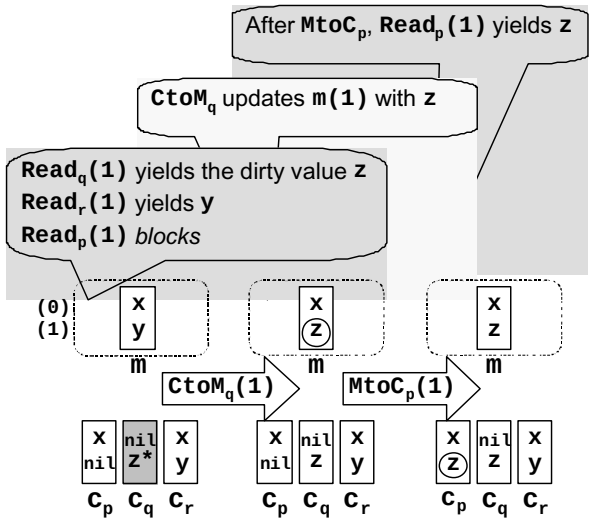
This memory is 'incoherent': different caches can have different data for the same address, so that reads and writes by different processors may see completely different data. Thus, it does not implement the `CoherentMemory` spec given earlier. However, after a $\texttt{Barrier}_p$, $\texttt{c}_p$ is guaranteed to agree with `m` until the next time `m` changes or `p` does a `Write`.[5] There are commercial machines whose memory systems have essentially this spec.[6] Others have explored similar specs.[7]

Here is a simple example that shows the contents of two addresses `0` and `1` in `m` and in three processors `p`, `q`, and `r`. A dirty value is marked with a `*`, and circles mark values that have changed. Initially $\texttt{Read}_q\texttt{(1)}$ yields the dirty value `z`, $\texttt{Read}_r\texttt{(1)}$ yields `y`, and $\texttt{Read}_p\texttt{(1)}$ blocks because $\texttt{c}_p\texttt{(1)}$ is `nil`. After the $\texttt{CtoM}_q$ the global location `m(1)` has been updated with `z`. After the $\texttt{MtoC}_p$, $\texttt{Read}_p\texttt{(1)}$ yields `z`. One way to ensure that the $\texttt{CtoM}_q$ and $\texttt{MtoC}_p$ actions happen before the $\texttt{Read}_p\texttt{(1)}$ is to do $\texttt{Barrier}_q$ followed by $\texttt{Barrier}_p$ between the $\texttt{Write}_q\texttt{(1)}$ that makes `z` dirty in $\texttt{c}_q$ and the $\texttt{Read}_p\texttt{(1)}$.

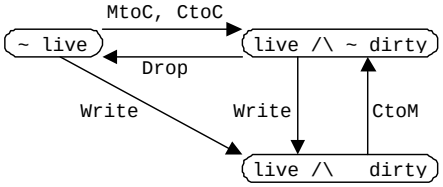
[5] An alternative version of `Barrier` has the guard `~ live`$_p$ `\/ (`c_p `= m)`; this is equivalent to the current $\texttt{Barrier}_p$ followed by an optional $\texttt{MtoC}_p$. You might think that it's better because it avoids a copy from `m` to $\texttt{c}_p$ in case they already agree. But this is a spec, not an implementation, and the change doesn't affect its external behavior.

[6] Digital Equipment Corporation, *Alpha Architecture Handbook*, 1992. IBM, *The PowerPC Architecture*, Morgan Kaufmann, 1994.

[7] Gharachorloo, K., et al., Memory consistency and event ordering in scalable shared-memory multiprocessors, *Proc. 17th Symposium on Computer Architecture*, 1990, pp 15-26. Gibbons, P. and Merritt, M., Specifying nonblocking shared memories, *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp 158-168.



Here are the possible transitions of `IncoherentMemory` for a given address. This kind of state transition picture is the standard way to describe cache algorithms in the literature; see pages 664-5 of Hennessy and Patterson, for example.



This is the weakest shared-memory spec that seems likely to be useful in practice. But perhaps it is too weak. Why do we introduce this messy incoherent memory? Wouldn't we be much better off with the simple and familiar coherent memory? There are two reasons to prefer `IncoherentMemory`.

- Code for `IncoherentMemory` can run faster — there is more locality and less communication. As we will see later in `ERxternalLocks`, software can batch the communication that is needed to make a coherent memory out of `IncoherentMemory`.
- Even `CoherentMemory` is tricky to use when there are concurrent clients. Experience has shown that it's necessary to have wizards to package it so that ordinary programmers can use it safely. This packaging takes the form of rules for writing concurrent programs and procedures that encapsulate references to shared memory. We studied these rules in handout 14 on practical concurrency, under the name 'easy concurrency'. The two most common examples are:

  Mutual exclusion / critical sections / monitors, which ensure that a number of references to shared memory can be done without interference, just as in a sequential program. Reader/writer locks are an important variation.

  Producer-consumer buffers.

For the ordinary programmer only the simplicity of the package is important, not the subtlety of its code. We need a smarter wizard to package `IncoherentMemory`, but the result is as simple to use as the packaged `CoherentMemory`.

Specifying legal histories directly

It's common in the literature to write the specs `CoherentMemory` and `IncoherentMemory` explicitly in terms of legal sequences of references in each processor, rather than as state machines (see the references in the previous section). We digress briefly to explain this approach, which is similar to what we did to specify concurrent transactions in handout 20.

For `CoherentMemory`^{LH}, there must be a total ordering of *all* the `Read`$_p$ and `Write`$_p$`(v)` actions done by the processors (for all the addresses) that

- respects the order at each `p`, and
- such that for each `Read` and closest preceding `Write(v)`, the `Read` returns `v`.

For `IncoherentMemory`^{LH}, *for each address separately* there must be a total ordering of the `Read`$_p$, `Write`$_p$, and `Barrier`$_p$ actions done by the processors that has the same properties. `IncoherentMemory` is weaker than `CoherentMemory` because it allows references to different addresses to be ordered differently. If there were only one address and no other communication (so that you couldn't see the relative ordering of the operations), you couldn't tell the difference between the two specs. A real barrier operation usually does a `Barrier` for every address, and thus forces all the references before it at a given processor to precede all the references after it.

It's not hard to show that `CoherentMemory`^{LH} is equivalent to `CoherentMemory`. It's less obvious that `IncoherentMemory`^{LH} is almost equivalent to `IncoherentMemory`. There's more to this spec than meets the eye, because it doesn't say anything about how the chosen ordering is related to the real times at which different processors do their operations. Actually it is somewhat more permissive than `IncoherentMemory`. For example, it allows the following history

- Initially `x=1`, `y=1`.
- Processor `p` reads `4` from `x`, then writes `8` to `y`.
- Processor `q` reads `8` from `y`, then writes `4` to `x`.

For `x` we have the ordering `Write`$_q$`(4); Read`$_p$, and for `y` the ordering `Write`$_p$`(8); Read`$_q$.

We can rule out this kind of predicting the future by observing that the processors make their references in some total order in real time, and requiring that a suitable ordering exist for the references in each prefix of this real time order. With this restriction, the two versions of `IncoherentMemory`^{LH} and `IncoherentMemory` are equivalent. But the restriction may not be an

improvement, since it's conceivable that a processor might be able to predict the future in this way by speculative execution. In any case, the memory spec for the Alpha is in fact IncoherentMemory[LH] and allows this freedom.

Coding coherent memory

We give a sequence of refinements that implement CoherentMemory and are successively more practical: GlobalImpl, Current Caches, and ExclusiveLocks. Then we give a different kind of code that is based on IncoherentMemory.

Global code

Now we give code for CoherentMemory. We obtain it simply by strengthening the guards on the operations of IncoherentMemory (omitting Barrier, which we don't need). This code is not practical, however, because the guards involve checking global state, not just the state of a single processor. This module, like later ones, maintains the invariant Inv3 that an address is dirty in at most one cache; this is necessary for the abstraction function to make sense. Note that the definition of Current says that the cache agrees with the abstract memory.

We show only the code that differs from IncoherentMemory, boxing the new parts.

```
MODULE GlobalImpl [P, A, V] EXPORT Read, Write =  % implements CoherentMemory

TYPE ...                                          % as in IncoherentMemory
VAR  ...

% ABSTRACTION: CoherentMemory.m = (Clean() => m [*] {p | dirty_p | c_p}.choose)

% INVARIANT Inv3: {p | dirty_p}.size <= 1         % dirty in at most one cache

APROC Read_p -> D = << Current_p  => RET c_p >>    % read only current data
APROC Write_p(d) =                                % Write maintains Inv3
    << Clean() \/ dirty_p  => c_p := d; dirty_p := true >>

FUNC Current_p =                                  % p's cache is current?
    RET c_p = (Clean() => m [*] {p | dirty_p | c_p}.choose)
FUNC Clean() = RET (ALL p | ~ dirty_p)            % all caches are clean?

% Same internal actions as IncoherentMemory.

END GlobalImpl
```

Notice that the guard on Read checks that the data in the processor's cache is current, that is, equals the value currently stored in the abstract memory. This requires finding the most recent value, which is either in the main memory (if no processor has a dirty value) or in some processor's cache (if a processor has a dirty value). The guard on Write ensures that a given address is dirty in at most one cache. These guards make it obvious that GlobalImpl implements CoherentMemory, but both require checking global state, so they are impractical to code directly.

Code in which caches are always current

We can't code the guards of GlobalImpl directly. In this section, we refine GlobalImpl a bit, replacing some (but not all) of the global tests. We carry this refinement further in the following sections. Our strategy for correctness is to always strengthen the guards in the actions, without changing the rest of the code. This makes it obvious that we simulate the previous module and that existing invariants hold. The only thing to check is that new invariants hold.

The main idea of CurrentCaches is to always keep the data in the caches current, so that we no longer need the Current guard on Read. In order to achieve this, we impose a guard on a write that allows it to happen only if no other processor has a cached copy. This is usually coded by having a write invalidate other cached copies before writing; in our code Write waits for Drop actions at all the other caches that are live. Note that Only implies the guard of GlobalImpl.Write because of Inv2 and Inv3, and Live implies the guard of GlobalImpl.Read because of Inv4. This makes it obvious that CurrentCaches implements GlobalImpl. CurrentCaches uses the non-local functions Clean and Only, but it eliminates Current. This is progress, because Read, the most common action, now has a local guard, and because Clean and Only just test Live and dirty, which is much simpler than Current's comparison of c_p with m.

As usual, the parts not shown are the same as in the last module, GlobalImpl.

```
MODULE CurrentCaches ... =                        % implements GlobalImpl

TYPE ...                                          % as in IncoherentMemory
VAR  ...

% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.

% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)  % data in caches is current

...

FUNC Only_p -> Bool = RET {p' | Live_p'} <= {p}   % appears at most in p's cache

APROC Read_p -> D = << Live_p  => RET c_p >>       % read locally; OK by Inv4
APROC Write_p(d) =                                % write locally the only copy
    << Only_p  => c_p := d; dirty_p := true >>

...

APROC MtoC_p = << Clean()  => c_p := m >>          guard maintains Inv4
...

END CurrentCaches
```

Code using exclusive locks

The next code refines CurrentCaches by introducing an exclusive (write) lock with a Free test and Acquire and Release actions. A writer must hold the lock on an object while it writes, but a reader need not hold any lock (Live acts as a read lock according to Inv6). Thus, multiple readers can read in parallel, but only one writer can write at a time, and only if there are no concurrent readers. This means that before a write can happen at p, all other processors must

drop their copies; making this happen is called 'invalidation'. The code ensures that while a processor holds a lock, no other cache has a copy of the locked object. It uses the non-local functions `Clean` and `Free`, but everything else is local. Again, the guards are stronger than those in `CurrentCaches`, so it's obvious that `ExclusiveLocks0` implements `CurrentCaches`. We show the changes from `CurrentCaches`.

```
MODULE ExclusiveLocks0 ... =                          % implements CurrentCaches

TYPE ...                                              % as in IncoherentMemory
VAR  ...
     lock        :  P -> Bool := {*->false}           % p has lock on cache?

% ABSTRACTION to CurrentCaches: Identity on m, c, and dirty.

% INVARIANT Inv5: {p | lock_p}.size <= 1              % lock is exclusive
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)         % locked data is only copy

...

APROC Write_p(d) =                                    % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>
...

FUNC Free() -> Bool = RET (ALL p | ~ lock_p)          % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p  [] SKIP OD

APROC MtoC_p =                                        % guard maintains Inv4, Inv6
     << Clean() /\  (lock_p \/ Free()) => c_p := m >>
APROC CtoC_p,p' =                                     % guard maintains Inv6
     << Free()  /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>

APROC Acquire_p = << Free() /\ Only_p => lock_p :=true >> % exclusive lock is on cache
APROC Release_p = << lock_p := false >>               % release at any time

...

END ExclusiveLocks0
```

Note that this all works even in the presence of cache-to-cache copying of dirty data; a cache can be dirty without being locked. A strategy that allows such copying is called *update-based*. The usual code broadcasts (on the bus) every write to a shared location. That is, it combines with each `Write`$_p$ a `CtoC`$_{p,\ p'}$ for each live `p'`. If this is done atomically, we don't need the `Only`$_p$ in `Acquire`$_p$ This is good if for each write of a shared location, the average number of reads on a different processor is near 1. It's bad if this average is much less than 1, since then each read that goes faster is paid for with many bus cycles wasted on updates.

It's possible to combine updates and invalidation. They you have to decide when to update and when to invalidate. It's possible to make this choice in a way that's within a factor of two of an

optimal algorithm that knows the future pattern of references.[8] The rule is to keep updating until the accumulated cost of updates equals the cost of a read miss, and then invalidate.

Both `Read` and `Write` now do only local tests, which is good since they are supposed to be the most common actions. The remaining global tests are the `Only` test in `Acquire`, the `Clean` test in `MtoC`, and the `Free` tests in `Acquire`, `MtoC`, and `CtoC`. In hardware these are most commonly coded by snooping on a bus. A processor can broadcast on the bus to check that:

- No one else has a copy (`Only`).
- No one has a dirty copy (`Clean`).
- No one has a lock (`Free`).

It's called 'snooping' because these operations always go along with transfers between cache and memory (except for `Acquire`), so no extra bus cycles are need to give every processor on the bus a chance to see them.

For this to work, another processor that sees the test must either abandon its copy or lock, or signal `false`. The `false` signals are usually generated at exactly the same time by all the processors and combined by a simple 'or' operation. The processor can also request that the others relinquish their locks or copies; this is called 'invalidating'. Relinquishing a dirty copy means first writing it back to memory, whereas relinquishing a non-dirty copy means just dropping it from the cache. Sometimes the same broadcast is used to invalidate the old copies and update the caches with new copies, although our code breaks this down into separate `Drop`, `Write`, and `CtoC` actions.

Keeping dirty data locked

In the next module, we eliminate the cache-to-cache copying of dirty data; that is, we eliminate updates on writes of shared locations. We modify `ExclusiveLocks` so that locks are held longer, until data is no longer dirty. Besides the delayed lock release, the only significant change is in the guard of `MtoC`. Now data can only be loaded into a cache `p` if it is not dirty in `p` and is not locked elsewhere; together, these facts imply that the data item is clean, so we no longer need the global `Clean` test.

```
MODULE ExclusiveLocks ... =                           % implements ExclusiveLocks0

TYPE ...                                              % as in ExclusiveLocks0
VAR  ...

% ABSTRACTION to ExclusiveLocks0: Identity on m, c, dirty, and lock.

% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)       % dirty data is locked

...

APROC MtoC_p =                                        % guard implies Clean()
     << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
```

[8] A. Karlin et al, Competitive snoopy caching. *Algorithmica* **3**, 1 (1988), pp 79-119.

```
APROC Release_p = << ~ dirty_p => lock_p := false >>      % don't release if dirty
...
END ExclusiveLocks
```

For completeness, we give all the code for `ExclusiveLocks`, since there have been so many incremental changes. The non-local operations are boxed.

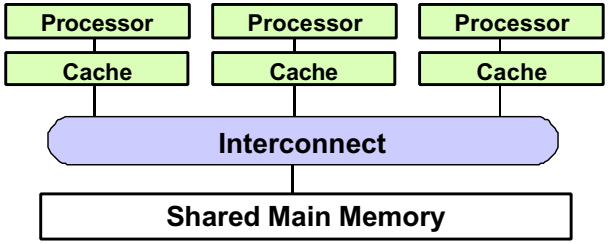
```
MODULE ExclusiveLocks[P,A,V] EXPORT Read,Write =  % implements CoherentMemory

TYPE M = D                                        % Memory
     C = P -> (D + Null)                          % Cache

VAR  m           :  CoherentMemory.M              % main memory
     c           := C{* -> nil}                   % local caches
     dirty       :  P -> Bool := {*->false}       % dirty flags
     lock        :  P -> Bool := {*->false}       % p has lock on cache?

% ABSTRACTION to ExclusiveLocks: Identity on m, c, dirty, and lock.

% INVARIANT Inv1: (ALL p | c!p)                   % every processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)    % dirty data is in the cache

% INVARIANT Inv3: {p | dirty_p}.size <= 1         % dirty in at most one cache
% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)  % data in caches is current
% INVARIANT Inv5: {p | lock_p}.size <= 1          % lock is exclusive
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)     % locked data is only copy
% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)    % dirty data is locked

APROC Read_p -> D = << Live_p => RET c_p >>       % read locally; OK by Inv4
APROC Write_p(d) =                                % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>

FUNC Live_p -> Bool = RET (c_p # nil)
FUNC Only_p -> Bool = RET {p' | Live_p'} <= {p}   % appears at most in p's cache?
FUNC Free() -> Bool = RET (ALL p | ~ lock_p)      % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p [] SKIP OD

APROC MtoC_p =                                    % guard implies Clean()
     << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >> % copy cache to memory.
APROC CtoC_p,p' =                                 % guard maintains Inv6
     << Free() /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>
APROC Drop_p = << ~ dirty_p => c_p := nil >>       % drop clean data from cache

APROC Acquire_p = << Free() /\ Only_p => lock_p :=true >> % exclusive lock is on cache
APROC Release_p = << ~ dirty_p => lock_p := false >>      % don't release if dirty

END ExclusiveLocks
```


Practical code

The remaining global tests are the `Only` test in the guard of `Acquire`, and the `Free` tests in the guards of `Acquire`, `MtoC` and `CtoC`. There are many ways to code them. Here are a few:

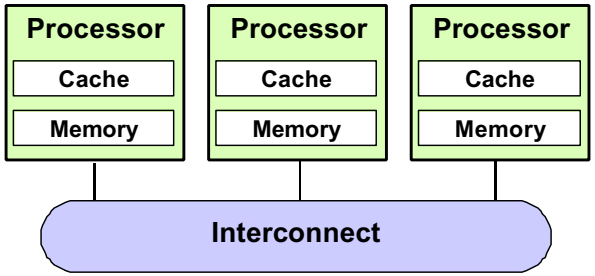
- Snooping on the bus, as described above. This is only practical when you have a cheap synchronous broadcast, that is, in a bus-based shared memory multiprocessor. The shared bus limits the maximum performance, so typically such systems are not built with more than about 8 processors.
- Directory-based: Keep a "directory", usually associated with main memory, containing information about where locks and copies are currently located. To check `Free`, a processor need only interact with the directory. To check `Only`, the same strategy can be used; however, there is a difficulty if cache-to-cache copying is permitted—the directory must be informed when such copying occurs. For this reason, directory-based code usually eliminates cache-to-cache copying entirely. So far, there's no need for broadcast. To acquire a lock, the directory may need to communicate with other caches to get them to relinquish locks and copies. This can be done by broadcast, but usually the directory keeps track of all the live processors and sends a message to each one.

  These schemes, both snooping and directory, are based on a model in which all the processors have uniform access to the shared memory.



  The directory technique extends to large-scale multiprocessor systems like Flash and Alewife, distributed shared memory, and locks in clusters[9], in which the memory is attached to processors. When the abstraction is memory rather than files, these systems are often called 'non-uniform memory access', or NUMA, systems.

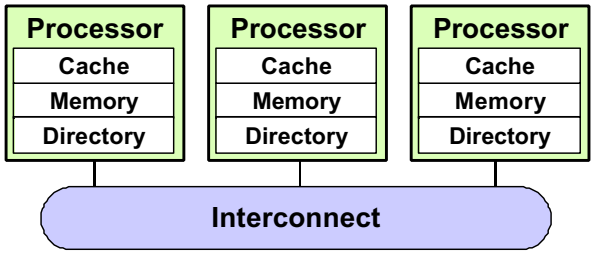
[9] Kronenberg, N. et al, The VAXCluster concept: An overview of a distributed system, *Digital Technical Journal* **1**, 3 (1987), pp 7-21.



The directory itself can be distributed by defining a 'home' location for each address that stores the directory information for that address. This is inefficient if that address turns out to be referenced mainly by other processors. To make the directory's distribution adapt better to usage, store the directory information for an address in a 'master' processor for that address, rather than in the home processor. The master can change to track the usage, but the home processor always remembers the master. Thus:

```
FUNC Home(a) -> P = …                                  % some fixed algorithm
VAR  master: P -> A -> P                               % master(p) is partial
     copies: P -> A -> SET P                           % defined only at the master
     locker: P -> A -> P                               % defined only at the master
INVARIANT (ALL a, p, p' |
            master(Home(a))!a                          % master is defined at a's home P,
          /\ master(p)!a /\ master(p')!a ==>           % where it's defined, it's the same
               master(p)(a) = master(p')(a)
          /\ copies!p = (p = master(Home(a))(a)) )     % and copies is defined only at master
```

The `Home` function is often a hash of `a`; it's possible to change the hash function, but if this is not done atomically it must be done very carefully, because `Home` will be different at different processors and the invariants must hold for all the different `Home`'s.



- Hierarchical: Partition the processors into sets, and maintain a directory for each set. The main directory attached to main memory keeps track of which processor sets have copies or locks; the directory for each set keeps track of which processors in the set have copies or locks. The hierarchy may have more levels, with the processor sets further subdivided, as in Flash.

There are many issues for high-performance code: communication cost, bandwidth into the cache into tag store, interleaving, and deadlock. The references at the start of this handout go into a lot of detail.

Purely software code is also possible. This form of DSM makes `V` be a whole virtual memory page and uses page faults to catch memory operations that require software intervention, while allowing those that can be satisfied locally to run at full speed. A live page is mapped, read-only unless it is dirty; a page that isn't live isn't mapped.[10]

Code based on `IncoherentMemory`

Next we consider a different kind of code for `CoherentMemory` that runs on top of `IncoherentMemory`. Coherence is guaranteed using an external read/write locking discipline. This is an example of an important general strategy—using weaker memory together with a programming discipline to guarantee strong coherence.

The code uses read/write locks, as defined earlier in the course, one per data item. There is a module $\texttt{ExternalLocks}_p$ for each processor `p`, which receives external `Read` and `Write` requests, obtains the needed locks, and invokes low-level `Read`, `Write`, and `Barrier` operations on the underlying `IncoherentMemory` memory. The composition of these pieces implements `CoherentMemory`. We give the code for $\texttt{ExternalLocks}_p$.

```
MODULE ExternalLocks_p [A, V] EXPORT Read, Write =    % implements CoherentMemory

% ReadAcquire_p acquires a read lock for processor p.
% Similarly for ReadRelease, WriteAcquire, WriteRelease

PROC Read_p =
    ReadAcquire_p; Barrier_p; VAR d| d := IncoherentMemory.Read_p; ReadRelease_p; RET d

PROC Write_p(d) = WriteAcquire_p; IncoherentMemory.Write_p(d); Barrier_p; WriteRelease_p

END ExternalLocks_p
```

This code does not satisfy all the invariants of `CurrentCaches` and its code. In particular, the data in caches is not always current, as stated in `Inv4`. It is only guaranteed to be current if it is read-locked, or if it is write-locked and dirty.

Invariants `Inv1`, `Inv2`, and `Inv3` are still satisfied. Invariants `Inv5` and `Inv6` no longer apply because the lock discipline is completely different; in particular, a locked copy need not be the only copy of an item. Let `wLockPs` be the set of processors that have a write-lock, and `rLockPs` be those with a read-lock.

We thus have `Inv1-3`, and new `Inv4a-Inv7a` that replace `Inv4-Inv7`:

```
% INVARIANT Inv4a:                                    % Data is current
    (ALL p | dirty_p \/ (p IN rLockPs /\ Live_p) ==> Current_p())
```

[10] K. Li and P. Hudak, Memory coherence in shared virtual memory systems, *ACM Transactions on Computer Systems* **7**, 4 (Nov 1989), pp 321-359.

```
% INVARIANT Inv5a:                                          % Write lock is exclusive.
     wLockPs.size <= 1

% INVARIANT Inv6a:                                          % Write lock excludes read locks.
     wLockPs # {} ==> rLockPs = {}

% INVARIANT Inv7a: (ALL p | dirty_p ==> p IN wLockPs)       % dirty data is write-locked
```

With these invariants, the identity abstraction to `GlobalImpl` works:

`% ABSTRACTION` to `GlobalImpl:` Identity on `m, c,` and `dirty.`

We note some differences between `ExternalLocks` and `ExclusiveLocks`, which also uses exclusive locks for writing:

- In `ExclusiveLocks`, `Read` can always proceed if there is a cache copy. In `ExternalLocks`, `Read` has a stronger guard in `ReadAcquire` (requiring a read lock).
- In `ExclusiveLocks`, `MtoC` checks that no other processor has a lock on the item. In `ExternalLocks`, an `MtoC` can occur as long as it doesn't overwrite dirty writes.
- In `ExternalLocks`, the guard for `Acquire` only involves lock conflicts, and does not check `Only`. (In fact, `ExternalLocks` doesn't use `Only` at all.)
- Additional `Barrier` actions are required in `ExternalLocks`.
- In `ExclusiveLocks`, the data in the cache is always current. In `ExternalLocks`, it is only guaranteed to be current for read-lock holders, and for write-lock holders who have already written.

In practice we don't surround every read and write with `Acquire` and `Release`. Instead, we take advantage of the rules for easy concurrency and rely on the fact that any reference to a shared variable must be in a critical section, surrounded by `Acquire` and `Release` of the lock that protects it. All we need to add is a `Barrier` at the beginning of the critical section, after the `Acquire`, and another at the end, before the `Release`. Sometimes people build these barrier actions into the acquire and release actions; this is called 'release consistency'.

Note—here we give up the efficiency of holding the lock until someone else needs it.

Remarks

Costs of incoherent memory

`IncoherentMemory` allows a multiprocessor shared memory to respond to `Read` and `Write` actions without any interprocessor communication. Furthermore, these actions only require communication between a processor and the global memory when a processor reads from an address that isn't in its cache. The expensive operation in this spec is `Barrier`, since the sequence $\texttt{Write}_p$; $\texttt{Barrier}_p$; $\texttt{Barrier}_q$; $\texttt{Read}_q$ requires the value written by `p` to be communicated to `q`. In most code `Barrier` is even more expensive because it acts on all addresses at once. This means that roughly speaking there must be at least enough

communication to record globally every address that `p` wrote before the $\texttt{Barrier}_p$, and to drop from `p`'s cache every address that is globally recorded as dirty.

Read-modify-write operations

Although this isn't strictly necessary, all current codes have additional external actions that make it easier to program mutual exclusion. These usually take the form of some kind of atomic read-modify-write operation, for example an atomic swap or compare-and-swap of a register value and a memory value. A currently popular scheme is two actions: `ReadLinked(a)` and `WriteConditional(a)`, with the property that if any other processor writes to `a` between a $\texttt{ReadLinked}_p\texttt{(a)}$ and the next $\texttt{WriteConditional}_p\texttt{(a)}$, the `WriteConditional` leaves the memory unchanged and returns an indication of failure. The effect is that if the `WriteConditional` succeeds, the entire sequence is an atomic read-modify-write from the viewpoint of another processor, and if it fails the sequence is a `SKIP`. Compare-and-swap is obviously a special case; it's useful to know this because something as strong as compare-and-swap is needed to program wait-free synchronization using a shared memory. Of course these operations also incur communication costs, at least if the address `a` is shared.

We have shown that a program that touches shared memory only inside a critical section cannot distinguish memory that satisfies `IncoherentMemory` from memory that satisfies the serial spec `CoherentMemory`. This is not the only way to use `IncoherentMemory`, however. It is possible to program other standard idioms, such as producer-consumer buffers, without relying on mutual exclusion. We leave these programs as an exercise for the reader.

Caching as easy concurrency

We developed the coherent caching code by evolving from the obviously correct `GlobalImpl` to code that has no global operations except to acquire locks. Another way to look at it is that coherent caching is just a variation on easy concurrency. Each `Read` or `Write` touches a shared variable and therefore must be done with a lock held, but there are no bigger atomic operations. The read lock is `Live` and the write lock is `lock`. In order to avoid the overhead of acquiring and releasing a lock on every memory operation, we use the optimization of holding onto a lock until some other cache needs it.

Write buffering

Hardware caches, especially the 'level 1' caches closest to the processor, usually come in two parts, called the cache and the write buffer. The latter holds dirty data temporarily before it's written back to the memory (or the level 2 cache in most modern systems). It is small and optimized for high write bandwidth, and for combining writes to the same cache block that happen close together in time into a single write of the entire line.

Invalidation

All caching systems have some provision for invalidating cache entries. A system that implements `CoherentMemory` usually must invalidate a cache entry that is written on another processor. The invalidation must happen before any read that follows the write touches the entry. Many systems, however, provide less coherence. For example, NFS simply times out cache

entries; this implements `IncoherentMemory`, with the clumsy property that the only way to code `Barrier` is to wait for the timeout interval. The web does caching in client browsers and also in proxies, and it also does invalidation by timeout. A web page can set the timeout interval, though not all caches respect this setting. The Internet caches the result of DNS lookups (that is, the IP addresses of DNS names) and of ARP lookups (that is, the LAN address of an IP address). These entries are timed out; a client can also discard an entry that doesn't seem to be working. The Internet also caches routing information, which is explicitly updated by periodic OSPF packets.

Think about what it would cost to make all these loosely coherent schemes coherent, and whether it would be worth it.

Locality and granularity

Caching works because the patterns of memory references exhibit locality. There are two kinds of locality.

- Temporal locality: if you reference an address, you are likely to reference it again in the near future, so it's worth keeping that item in the cache.
- Spatial locality: if you reference an address, you are likely to reference a neighboring address in the near future. This makes it worthwhile to transfer a large block of data to the cache, since the overhead of a miss is only paid once. Large blocks do have two drawbacks: they consume more bandwidth, and they introduce or increase 'false sharing'. A whole block has to be invalidated whenever any part of it is written, and if you are only reading a different part, the invalidation makes for extra work.

Both temporal and spatial locality can be improved by restructuring the program, and often this restructuring can be done automatically. For instance, it's possible to rearrange the basic blocks of a program based on traces of program execution to put blocks that normally follow each other in traces in the same cache line or virtual memory page.

Distributed file systems

A distributed file system does caching which is logically identical to the caching that a memory system does. There are some practical differences:

- A DFS is usually built without any hardware support, whereas most DSM's depend at least on the virtual memory system to detect misses while letting hits run at full local memory speed, and perhaps on much more hardware support, as in Flash.
- A DFS must deal with failures, whereas a DSM usually crashes a program that is sharing memory with another program that fails.
- A DFS usually must scale better, to hundreds or thousands of nodes.
- A DFS has a wider choice of granularity: whole files, or a wide range of block sizes within files.

.