Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 44 November 17, 1993

# Distributed Transactions

Making transactions work in a distributed setting requires solving two problems. First, we need to deal with partial failures: some but not all sites may fail, and we must ensure that a transaction commits only if all its updates are recorded stably at all sites involved in the transaction. Second, the different sites need to coordinate in synchronizing concurrent transactions. Below, we describe solutions to the *atomic commitment* problem, which abstracts the problem of ensuring that different sites reach consistent conclusions about the outcome of a transaction. Then we describe interactions between atomic commitment and crash recovery. Finally, we discuss issues involved in coordinating distributed synchronization.

## 1 Atomic Commitment

Atomic commitment is an agreement problem, similar to consensus. It differs in the validity condition. For consensus, any allowed value can be chosen as the outcome of consensus. For atomic commitment, we permit an outcome of "commit" only if every participant allows "commit"; otherwise the outcome must be "abort".

Here is the specification. Notice that it chooses the outcome in a background thread, so that no prophecy variable should be needed to prove an implementation correct. Also notice that the names of the processes involved in the protocol are part of the specification; this is necessary to be able to specify the validity condition.

```
MODULE AtomicCommit [ P ] =                  % P is process name

TYPE D = ENUM[commit,abort]                  % allowed values and outcomes

VAR
   consensus   :  (D + Null) := nil
   outcome     :  P -> (D + Null) := {*->nil}
   allowed     :  P -> (D + Null) := {*->nil}
```

```
% Agreement:  ( EXISTS d | (ALL p | outcome(p) IN {d, nil}))
% Validity:   ( ALL p    | outcome(p) = commit ==>
                                (ALL p | allowed(p) = commit))

APROC Allow(p, d) = << allowed(p) = nil => allowed(p) := d [*] HAVOC >>

THREAD Agree() =
    <<    (ALL p | allowed(p) = commit) /\ consensus = nil =>
                                 consensus := commit
       [] consensus = abort => SKIP >>;
    DO << VAR p | outcome(p) = nil => outcome(p) := consensus >> OD

FUNC Outcome(p) -> (D + Null) = RET outcome(p)

% Failure corresponds to a crash of a participant or a lost
% or delayed message.  Abort is an allowable outcome only if one
% of these failures occurs.
APROC Failure() = << consensus = nil => consensus := abort >>

END AtomicCommit
```

A simple way to implement atomic commitment is to use consensus as a submodule to ensure agreement among the participants in a transaction. To ensure the validity condition, a "coordinator" first broadcasts a request for allowed values to all participants; only if all respond "commit" does the coordinator allow "commit" as a possible outcome.

```
MODULE Consensus = Consensus[ ACImpl.D ]

MODULE ACImpl [ P ] =

TYPE
   D        = ENUM[commit,abort]
   DN       = (D + Null)
   K        = ENUM[prepare, allowed]          % Kind of message
   M        = [k,                             % Message: kind,
                 p,                           % sender
                 dn ]                         % data

VAR
   % Stable.
   outcome     :  P -> DN := {*->nil}
   % Volatile.
```

```
   allowed     :  P -> DN := {*->nil}
   ms          :  SET M := {}               % Messages in transit

% Abstraction function:
%    outcome in the spec is taken directly from outcome in the implementation.
%    consensus in the spec is taken from outcome in the Consensus module.
%    allowed requires a history variable to remove the effects of forcing
%    a participant's allowed value to ``abort" when it responds to the
%    coordinator without having already recorded an allowed value.
%    (Forcing the participant to allow ``abort" in this case is an
%    indication that a failure must have happened if we assume that a
%    participant allows ``commit" before receiving a prepare message
%    unless it has crashed and aborted the transaction.)

APROC Allow(p, d) = << allowed(p) = nil => allowed(p) := d [*] SKIP >>

FUNC Outcome(p) -> DN = RET outcome(p)

THREAD Actions(p) =
   VAR
      m,                                    % Message
      Vs : P->DN  := {* -> nil}             % Collected allowed values
   |
   DO << % pick an enabled action and do it.

         % Crash
         Vs := {*->nil}; allowed(p) := nil

      [] % Coordinator:  request allowed values
         Send(prepare,p,nil)

      [] % Participant:  send response to request.
         % Forcing the allowed value to abort corresponds to a failure.
         << m := Receive(prepare);
                allowed(p) = nil => allowed(p) := abort;
                                    Send(allowed,p,abort)
            [*] Send(allowed,p,allowed(p)) >>

      [] % Coordinator:  receive allowed value
         m := Receive(allowed);
         Vs(m.p) := m.dn
```

```
    [] % Coordinator:  initiate consensus
       (ALL p | Vs(p) = commit) => Consensus.Allow(p,commit)

    [] % Any participant:  give up (only on failure of some sort)
       Consensus.Allow(p,abort)

    [] % Record p's outcome
       outcome(p) = nil => outcome(p) := Consensus.Outcome()
 >> OD

APROC Send(k,p,dn) = << ms := ms ++ M{k:=k,p:=p,dn:=dn} >>
APROC Receive(k) -> M = << VAR m | m IN ms /\ m.k = k => RET m >>
THREAD Lose() = << VAR m | m IN ms => ms := ms - {m} >>

END ACImpl
```

The usual implementation of ACImpl, called the *two-phase commit protocol* (not to be confused with two-phase locking), uses a very simple implementation for Consensus: the coordinator records his allowed value stably as the outcome, and all participants wait to hear the decision from the coordinator. If the coordinator crashes, there is no way for another process to take over as coordinator (unlike in Paxos, where any process can become the leader). Thus, crashes and network partitions can cause processes to block waiting to hear the outcome.

The Paxos protocol requires an extra round of messages compared to two-phase commit, but is more robust: if the leader crashes, the other processes can still reach a decision about the outcome. Using Paxos, a process is forced to block only if there is a network partition and the process is on the minority side of the partition.

Note that the implementation above, if based on Paxos as the implementation of Consensus, might seem to require an initial round of communication for the coordinator to collect allowed values, followed by two and a half rounds for Paxos itself—two extra rounds compared to two-phase commit. However, collecting allowed values can be done at the same time as the first round of Paxos (which collects the "votes" from earlier ballots), so that in fact a total of only two and a half rounds of communication are required.

## 2 Interactions With Crash Recovery

A protocol that implements atomic commitment alone isn't enough; we also need to modify the crash recovery procedure. In particular, on recovery from a crash, we can no longer abort all transactions that were active (not yet committed or aborted) at the time of the crash. A single site can no longer make a unilateral decision about the outcome of a transaction; instead, it must ensure that it and all the other sites involved in the transaction agree on the outcome.

There is one case in which a site can make a unilateral decision. If a site crashes and an active transaction is not yet prepared (i.e., the consensus protocol for committing the transaction has not yet started), then the site can abort the transaction on recovery – since it can then force an abort when the commit protocol runs. Thus, we divide transactions into four classes: committed, aborted, prepared, and active. Active transactions can be aborted after a crash, but prepared transactions must be left in the prepared state until the outcome of the consensus protocol is known. This means that their locks must be maintained and the appropriate recovery information must be kept to allow them to be either committed or aborted depending on the outcome of the consensus protocol.

# 3 Coordinating Synchronization

Simply requiring serializability at each site in a distributed transaction system is not enough, since the different sites could choose different serialization orders. To ensure that a single global serialization order exists, we need stronger constraints on the individual sites. We can capture these constraints in a specification. It turns out that there are many different specifications we could give, each of which corresponds to a different class of mutually compatible concurrency control methods (but where two concurrency control methods from two different classes may be incompatible). Here we illustrate one possible specification, which is appropriate for systems that use strict two-phase locking and other compatible concurrency control methods.

Strict two-phase locking is one of many methods that serializes transactions in the order in which they commit (which is the same as the order of their lockpoints). Our goal is to capture this constraint — that committed transactions are serializable in the order in which they commit — in a specification for individual sites in a distributed transaction system. This cannot be done directly, because commit decisions are made in a decentralized manner, so no single site knows the commit order. However, each site has some information about the global commit order. In particular, if a site hears that transaction $A$ has committed before it processes an operation for transaction $B$, then $B$ must follow $A$ in the global commit order (assuming that $B$ eventually commits). Given a site's local knowledge, there is a set of global commit orders consistent with its local knowledge (one of which must be the actual commit order). Thus, if a site ensures serializability in all possible commit orders consistent with its local knowledge, it is necessarily ensuring serializability in the global commit order.

We can capture this idea more precisely in the following specification. (Rather than giving all the details, we sketch how to modify the specification of concurrent transactions given earlier in the semester.)

- Add a new operation `InformCommit` to tell a site that a transaction has committed. This operation is called when the site learns the outcome of the atomic commitment protocol for the transaction.

- Keep track of a partial order `precedes` on transactions, which should record that $A$ `precedes` $B$ whenever the `InformCommit` procedure for $A$ happens before an action is done for $B$. This can be done either by keeping a history variable with all external operations recorded (and defining `precedes` as a function on the history variable), or by explicitly updating `precedes` whenever an action is performed for a transaction (by adding all pairs $(A, B)$ where $A$ is known to be committed and $B$ is performing the action).

- Change the constraint `Serializable` in the invariant in the specification to require serializability in all total orders consistent with `precedes`, rather than in any externally consistent order. (Notice that an order consistent with `precedes` is also externally consistent.)

It is easy to show that the order in which transactions commit is one total order consistent with `precedes`; thus, if every site ensures serializability in every total order consistent with its local `precedes` order, it follows that the global commit order can be used as a global serialization order.