27. Distributed Transactions

In this handout we study the problem of doing a transaction (that is, an atomic action) that involves actions at several different transaction systems, which we call the *resource managers* or RMs. The most obvious application is "distributed transactions": separate databases running on different computers. For example, we might want to transfer money from an account at Citibank to an account at Wells Fargo. Each bank runs its own transaction system, but we still want the entire transfer to be atomic. More generally, however, it is good to be able to build up a system recursively out of smaller parts, rather than designing the whole thing as a single unit. The different parts can have different code, and the big system can be built even though it wasn't thought of when the smaller ones were designed. For example, we might want to run a transaction that updates some data in a database system and some other data in a file system.

Specs

We have to solve two problems: composing the separate RMs so that they can do a joint action atomically, and dealing with partial failures. Composition doesn't require any changes in the spec of the RMs; two RMs that implement the `SequentialTr` spec in handout 19 can jointly commit a transaction if some third agent keeps track of the transaction and tells them both to commit. Partial failures do require changes in the resource manager spec. In addition, they require, or at least strongly suggest, changes in the client spec. We consider the latter first.

The client spec

In the code we have in mind, the client may be invoking `Do` actions at several RMs. If one of them fails, the transaction will eventually abort rather than committing. In the meantime, however, the client may be able to complete `Do` actions at other RMs, since we don't want each RM to have to verify that no other RM has failed before performing a `Do`. In fact, the client may itself be running on several machines, and may be invoking several `Do`'s concurrently. So the spec should say that the transaction can't commit after a failure, and can abort any time after a failure, but need not abort until the client tries to commit. Furthermore, after a failure some `Do` actions may report `crashed`, and others, including some later ones, may succeed.

We express this by adding another value `failing` to the phase. A crash sets the phase to `failing`, which enables an internal `CrashAbort` action that aborts the transaction. In the meantime a `Do` can either succeed or raise `crashed`.

```
CLASS DistSeqTr [
    V,                                              % Value of an action
    S WITH { s0: ()->S }                            % State
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                        % Action

VAR  ss         := S.s0()                           % Stable State
     vs         := S.s0()                           % Volatile State
     ph         : ENUM[idle, run, failing] := idle  % PHase (volatile)
```



```
APROC Begin() = << Abort(); ph := run >>                % aborts any current trans.

APROC Do(a) -> V RAISES {crashed} = <<                  % non-deterministic if failing!
    IF  ph # idle => VAR v | (v, vs) := a(vs); RET v
    []  ph # run  => RAISE crashed
    FI >>

APROC Commit() RAISES {crashed} =
    << IF ph = run => ss := vs; ph := idle [*] Abort(); RAISE crashed FI >>

PROC Abort () = << vs := ss, ph := idle >>
PROC Crash () = << ph := failing >>

THREAD CrashAbort() = DO << ph = failing => Abort() >> OD

END DistSeqTr
```

In a real system `Begin` plays the role of `New` for this class; it starts a new transaction and returns its transaction identifier `t`, which is an argument to every other routine. Transactions can commit or abort independently (subject to the constraints of concurrency control). We omit this complication. Dealing with it requires representing each transaction's state change independently in the spec, rather than just letting them all update `vs`. If the concurrency spec is 'any can commit', for example, `Do(t)` sees `vs = ss + actions(t)`, and `Commit(t)` does `ss := ss + actions(t)`.

Partial failures

When several RMs are involved in a transaction, they must agree about whether the transaction commits. Thus each transaction commit requires consensus among the RMs.

The code that implements transactions usually keeps the state of a transaction in volatile storage, and only guarantees to make it stable at commit time. This is important for efficiency, since stable storage writes are expensive. To do this with several RMs requires a RM action to make a transaction's state stable without committing it; this action is traditionally called `Prepare`. We can invoke `Prepare` on each RM, and if they all succeed, we can commit the transaction. Without `Prepare` we might commit the transaction, only to learn that some RM has failed and lost the transaction state.

`Prepare` is a formalization of the so-called write-ahead logging in the old `LogRecovery` or `LogAndCache` code in handout 19. This code does a `Prepare` implicitly, by forcing the log to stable storage before writing the commit record. It doesn't need a separate `Prepare` action because it has direct and exclusive access to the state, so that the sequential flow of control in `Commit` ensures that the state is stable before the transaction commits. For the same reason, it doesn't need separate actions to clean up the stable state; the sequential flow of `Commit` and `Crash` takes care of everything.

Once a RM is prepared, it must maintain the transaction state until it finds out whether the transaction committed or aborted. We study a design in which a separate 'coordinator' module is responsible for keeping track of all the RMs and telling them to commit or abort. Real systems sometimes allow the RMs to query the coordinator instead of, or in addition to, being told what to do, but we omit this minor variation.

We first give the spec for a RM (not including the coordinator). Since we want to be able to compose RMs repeatedly, we give it as a modification (*not* an implementation) of the `DistSeqTr` client spec; this spec is intended to be called by the coordinator, not by the client (though, as we shall see, some of the procedures can be called directly by the client as an optimization). The

change from DistSeqTr is the addition of the stable 'prepared state' ps, and a separate Prepare action between the last Do and Commit. A transaction is prepared if ps # nil. Note that Crash has no effect on a prepared transaction. Abort works on any transaction, prepared or not.

```
TYPE T = (Coord + Null)                                  % Transaction id; see below for Coord

CLASS RMTr [
    V,                                                   % Value of an action
    S WITH { s0: ()->S },                                % State
    RM
    ] EXPORT Begin, Do, Commit, Abort, Prepare, Crash =

TYPE A          =  S->(V, S)                             % Action

VAR  ss         := S.s0()                                % Stable State
     ps         :  (S + Null) := nil                     % Prepared State (stable)
     vs         := S.s0()                                % Volatile State
     ph         : ENUM[idle, run, failing] := idle       % PHase (volatile)
     rm                                                  % the RM that contains self
     t          := nil                                   % "transaction id"; just for invariants

% INVARIANT ps # nil  ==>  ph = idle

APROC Begin(t') = << ph := run; t := t' >>

APROC Do(a) -> V RAISES {crashed} = <<                   % non-deterministic if ph=failing
    IF   ph # idle => VAR v | (v, vs) := a(vs); RET v
    []   ph # run => RAISE crashed
    FI >>

APROC Prepare() RAISES {crashed} =                       % called by coordinator
    << IF ph = run => ps := vs; ph := idle [*] RAISE crashed >>

APROC Commit() = <<                                      % succeeds only if prepared
    IF ps # nil => ss := ps; ps := nil [*] SKIP FI >>

PROC Abort () = << vs := ss, ph := idle; ps := nil >>
PROC Crash () = << IF ps = nil => ph := failing [*] SKIP >>

THREAD CrashAbort() = DO << ph = failing => Abort() >> OD

END RMTr
```

The idea of this spec is that its client is the coordinator, which implements DistSeqTr using one or more copies of RMTr. As we shall see in detail later, the coordinator

passes Do directly through to the appropriate RMTr,

does some bookkeeping for Begin, and

earns its keep with Commit by first calling Prepare on each RMTr and then, if all these are successful, calling Commit on each RMTr.

Optimizations discussed below allow the client to call Do, and perhaps Begin, directly. The RMTr spec requires its client to call Prepare exactly once before Commit. Note that because Do raises crashed if ph # idle, it raises crashed after Prepare. This reflects the fact that it's an error to do any actions after Prepare, because they wouldn't appear in ps. A real system might handle

these variations somewhat differently, for instance by raising tooLate for a Do while the RM is prepared, but the differences are inessential.

We don't give code for this spec, since it is very similar to LogRecovery or LogAndCache. Like the old Commit, Prepare forces the log to stable storage; then it writes a prepared record (coding ps # nil) so that recovery will know not to abort the transaction. Commit to a prepared transaction writes a commit record and then applies the log or discards the undo's. Recovery rebuilds the volatile list of prepared transactions from the prepared records so that a later Commit or Abort knows what to do. Recovery must also restore the concurrency control state for prepared transactions; usually this means re-acquiring their locks. This is similar to the fact that recovery in LogRecovery must re-acquire the locks for undo actions; in that case the transaction is sure to abort, while here it might also commit.

Committing a transaction

We have not yet explained how to code DistSeqTr using several copies of RMTr. The basic idea is simple. A coordinator keeps track of all the RMs that are involved in the transaction (they are often called 'workers', 'participants', or 'slaves' in this story). Normally the coordinator is also one of the RMs, but as with Paxos, it's easier to explain what's going on by keeping the two functions entirely separate. When the client tells the coordinator to commit, the coordinator tells all the RMs to prepare. This succeeds if all the Prepare's return normally. Then the coordinator records stably that the transaction committed, returns success to the client, and tells all the RMs to commit.

If some RM has failed, its Prepare will raise crashed. In this case the coordinator raises crashed to the client and tells all the RMs to abort. A RM that is not prepared and doesn't hear from the coordinator can abort on its own. A RM that is prepared cannot abort on its own, but must hear from the coordinator whether the transaction has committed or aborted. Note that telling the RMs to commit or abort can be done in background; the fate of the transaction is decided at the coordinator.

The abstraction function from the states of the coordinator and the RMs to the state of DistSeqTr is simple. We make RMTr a class, so that the type RMTr refers to an instance. We call it R for short. The RM states are thus defined by the RMTr class (which is an index to the RMTrs table of instances that is the state of the class (see section 7 of handout 4 for an explanation of how classes work in Spec).

The spec's vs is the combination of all the RM vs values, where 'combination' is some way of assembling the complete state from the pieces on the various RMs. Most often the state is a function from variables to values (as in the Spec semantics) and the domains of these functions are disjoint on the different RMs. That is, the state space is partitioned among the RMs. Then the combination is the overlay of all the RMs' vs functions. Similarly, the spec's ss is the combination of the RMs' ss unless ph = committed, in which case any RM with a non-nil ps substitutes that.

We need to maintain the invariant that any R that is prepared is in rs, so that it will hear from the coordinator what it should do.

```
CLASS Coord [

TYPE R          =  RMTr                                  % instance name on an RM
     Ph         =  ENUM[idle, commit]                    % PHase
```

```
CONST AtoRM      :  A -> RM := ...                    % the RM that handles action a

VAR  ph                                               % ph and rs are stable
     rs          :  SET R                             % the slave RMs
     finish      :  Bool                              % outcome is decided; volatile

ABSTRACTION
% assuming a partitioned state space, with S as a function from state component names to values; + combines these
     DistSeqTr.vs = + : rs.vs
     DistSeqTr.ss = (ph # commit => + : rs.ss
                     [*]  + : (rs * (\ r | (r.ps # nil => r.ps [*] r.ss))) )

INVARIANT
        r :IN rs ==> r.coord = self                   % slaves know the coordinators's id
     /\ {r | r.coord = self /\ r.ps # nil} <= rs      % r prepared => in rs

APROC Begin() = << ph := idle; rs := {}; finish := false >>

PROC Do(a) -> V RAISES {crashed} =
% abstractly r.begin=SKIP and rs is not part of the abstract state, so abstractly this is an APROC
     IF   ph = idle => VAR r := AtoR(a) |
          IF r = nil =>
               r := R.new(); r.rm := AtoRM(a); r.begin(self); rs + := r
          [*] SKIP FI;
          r.do(a)
     [*]  RAISE crashed FI

PROC Commit() RAISES {crashed} =
     IF ph = idle => VAR rs' |
          ForAllRs(R.prepare) EXCEPT crashed => Abort(); RAISE crashed;
          ph := commit; finish := true
     [*]  Abort(); RAISE crashed FI

PROC Abort() = finish := true

THREAD Finish() = finish =>                           % tell all RMs what happened
% It's OK to do this in background, after returning the transaction outcome to the client
     ForAllRs((ph = commit => R.commit [*] R.abort));
     ph := idle; rs := {}                             % clean up state; can be deferred

PROC Crash() = finish := true                         % rs and ph are stable

FUNC AtoR(a) -> (R + Null) = VAR r :IN rs | r.rm = AtoRM(a) => RET r [*] RET nil
% If we've seen the RM for this action before, return its r; otherwise return nil

PROC ForAllRs(p: PROC R->() RAISES {crashed}) RAISES crashed =
% Apply p to every RM in rs
     VAR rs' := rs | DO VAR r :IN rs' | p(r); rs' - := {r} OD

END Coord
```

We have written this in the way that Spec makes most convenient, with a class for `RMTr` and a class for `Coord`. The coordinator's identity (of type `Coord`) identifies the transaction, and each `RMTr` instance keeps track of its coordinator; the first invariant in `Coord` says this. In a real system, there is a single transaction identifier `t` that labels both coordinator and slaves, and you identify a slave by the pair `(rm, t)` where `rm` is the recourse manager that hosts the slave. We earlier defined `T` to be short for `Coord`:

This entire algorithm is called "two-phase commit"; do not confuse it with two-phase locking. The first phase is the prepares (the write-ahead logging), the second the commits. The coordinator can use any algorithm it likes to record the commit or abort decision. However, once some RM is prepared, losing the commit/abort decision will leave that RM permanently in limbo, uncertain whether to commit or abort. For this reason, a high-availability transaction system should use a high-availability way of recording the commit. This means storing it in several places and using a consensus algorithm to get these places to agree.

For example, you could use the Paxos algorithm. It's convenient (though not necessary) to use the RMs as the agents and the coordinator as leader. In this case the query/report phase of Paxos can be combined with the prepares, so no extra messages are required for that. There is still one round of command/report messages, which is more expensive than the minimum, non-fault-tolerant consensus algorithm, in which the coordinator just records its decision. But using Paxos, a RM is forced to block only if there is a network partition and it is on the minority side of the partition.

In the theory literature this form of consensus is called the 'atomic commitment' problem. We can state the validity condition for atomic commitment as follows: A crash of any unprepared RM does `Allow(abort)`, and when the coordinator has heard that every RM is prepared it does `Allow(commit)`. You might think that consensus is trivial since at most one value is allowed. Unfortunately, this is not true because in general you don't know which value it is.

Most real transaction systems do not use fault-tolerant consensus to commit, but instead just let the coordinator record the result. In fact, when people say 'two-phase commit' they usually mean this form of consensus. The reason for this sloppiness is that usually the RMs are not replicated, and one of the RMs is the coordinator. If the coordinator fails or you can't communicate with it, all the data it handles is inaccessible until it is restored from tape. So the fact that the outcome of a few transactions is also inaccessible doesn't seem important. Once RMs are replicated, however, it becomes important to replicate the commit result as well. Otherwise that will be the weakest point in the system.

Bookkeeping

The explanation above gives short shrift to the details of making the coordinator efficient. In particular, how does the coordinator keep track of the RMs efficiently. This problem has three aspects: enumerating RMs, noticing failed RMs, and cleaning up. The first two are caused by wanting to allow clients to talk directly to RMs for everything except commit and abort.

Enumerating RMs

The first is simply finding out who the RMs are, since for a single transaction the client may be spread out over many processes, and it isn't efficient to funnel every request to a RM through the coordinator as this code does. The standard way to handle this is to arrange all the client processes that are part of a single transaction in a tree, and require that each client process report to its parent the RMs that it or its children have talked to. Then the client at the root of the tree will know about all the RMs, and it can either act as coordinator itself or give the coordinator this information. The danger of running the coordinator on the client, of course, is that it might fail and leave the RMs hanging.

Noticing failed RMs

The second is noticing that an RM has failed during a transaction. In the `SequentialTr` or `DistSeqTr` specs this is simple: each transaction has a `Begin` that sets `ph := run`, and a failure

sets ph to some other value. In the code, however, since again there may be lots of client processes cooperating in a single transaction, a client doesn't know the first time it talks to a RM, so it doesn't know when to call Begin on that RM. One way to handle this is for each client process to send Begin to the coordinator, which then calls Begin exactly once on each RM; this is what Coord does. This costs extra messages, however, An alternative is to eliminate Begin and instead have both Do and Prepare report to the client whether the transaction is new at that RM, that is, whether ph = idle before the action; this is equivalent to having the RM tell the client that it did a Begin along with the Do or Prepare. If a RM fails, it will forget this information (unless it's prepared, in which case the information is stable), so that a later client action will get another 'new' report. The client processes can then roll up all this information. If any RM reports 'new' more than once, it must have crashed.

To make this precise, each client process in a transaction counts the number of 'new' reports it has gotten from each RM (here C names the client processes):

```
VAR news        : C -> R -> Int := {* -> 0}
```

We add to the RM state a history variable lost which is true if the RM has failed and lost some of the client's state. This is what the client needs to detect, so we maintain the invariant (here clientPrs(t) is the set of client processes for t):

```
( ALL r | r.t = t /\ r.lost ==>
       r.ph = idle /\ r.ps = nil)
    \/ (+ : {c :IN clientPrs(t) || news(c)(r)}) > 1 ) )
```

After all the RMs have prepared, they all have r.ps # nil, so if anything is lost is shows up in the news count. The second disjunct says that across all the client processes that are running t, r has reported 'new' more than once, and therefore must have crashed during the transaction. As with enumerating the RMs, we collect this information from all the client processes before committing.

A variation on this scheme has each RM maintain an 'incarnation id' or 'crash count' which is different each time it recovers, and report this id to each Do and Prepare. Then any RM that is prepared and has reported more than one id must have failed during the transaction. Again, the RM doesn't know this, but the coordinator does.

Cleaning up

The third aspect of bookkeeping is making sure that all the RMs find out whether the transaction committed or aborted. Actually, only the prepared RMs really need to find out, since a RM that isn't prepared can just abort the transaction if it is left in the lurch. But the timeout for this may be long, so it's usually best to inform all the RMs if it's convenient.

There's no problem if the coordinator doesn't crash, since it's cheap to maintain a volatile rs, although it's expensive to maintain a stable rs as Coord does. If rs is volatile, however, then the coordinator won't know who the RMs are after a crash. If the coordinator remembers the outcome of a transaction indefinitely this is OK; it can wait for the RMs to query it for the outcome after a crash. The price is that it can never foorget the outcome, since it has no way of knowing when it's heard from all the RMs. We say that a T with any prepared RMs is "pending", because some RM is waiting to know the outcome. If the coordinator is to know when it's no longer pending, so that it can forget the outcome, it needs to know (a superset of) all the prepared RMs and to hear that each of them is no longer prepared but has heard the outcome and taken the proper action.

So the choices appear to be either to record rs stably before Prepare, in which case Coord can forget the outcome after all the RMs know it, at the price of an ack message from each RM, or to remember the outcome forever, in which case there's no need for acks; Coord does the former. Both look expensive. We can make remembering the outcome cheap, however, if we can *compute* it as a function Presumed of the transaction id t for any transaction that is pending.

The simplest way to use these ideas is to make Presumed(t) = aborted always, and to make rs stable at Commit, rather than at the first Prepare, by writing it into the commit record. This makes aborts cheap: it avoids an extra log write before Prepare and any acks for aborts. However, it still requires each RM to acknowledge the Commit to the coordinator before the coordinator can forget the outcome. This costs one message from each RM to the coordinator, in addition to the unavoidable message from the coordinator to the RM announcing the commit. Thus presumed abort optimizes aborts, which is stupid since aborts are rare.

The only way to avoid the acks on commit is to make Presumed(t) = committed. This is not straightforward, however, because now Presumed is not constant. Between the first Prepare and the commit, Presumed(t) = aborted because the outcome after a crash is aborted and there's no stable record of t, but once the transaction is committed the outcome is committed. This is no problem for t.Commit, which makes t explicit by setting ph := commit (that is, by writing a commit record in the log), but it means that by the time we forget the outcome (by discarding the commit record in the log) so that t becomes presumed, Presumed(t) must have changed to committed.

This will cost a stable write, and to make it cheap we batch it, so that lots of transactions change from presumed abort to presumed commit at the same time. The constraint on the batch is that if t aborted, Presumed(t) can't change until t is no longer pending.

Now comes the tricky part: after a crash we don't know whether an aborted transaction is pending or not, since we don't have rs. This means that we can't change Presumed(t) to committed for any t that was active and uncommitted at the time of the crash. That set of t's has to remain presumed abort forever. Here is a picture that shows one way that Presumed can change over time:

PC | PA-live | future
PresumeCommitted
PC | PC | PA-live | future
Crash
PC | PC | PA+*ph*=*c* | PA-live | future
PresumeCommitted
PC | PC | PA+*ph*=*c* | PC | PA-live |future

Note that after the crash, we have a permanent section of presumed-abort transactions, in which there might be some committed transactions whose outcome also has to be remembered forever. We can avoid the latter by making rs stable as part of the Commit, which is cheap. We can avoid the permanent PA section entirely by making rs stable before Prepare, which is not cheap. The following table shows the various cost tradeoffs.

	Commit, no crash	*Commit, crash*	*Abort, no crash*	*Abort, crash*
`Coord`				
Stable `rs`	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks
Presumed abort				
Volatile `rs` only	ph, ~~acks~~	ph, ~~acks~~	~~ph~~, ~~acks~~	~~ph~~, ~~acks~~
Stable `rs` on commit	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~acks~~	~~ph~~, ~~acks~~
Presumed commit				
Volatile `rs` only	~~ph~~, ~~acks~~	ph, ~~acks~~	~~ph~~, acks	ph, ~~acks~~
Stable `rs` on commit	~~ph~~, ~~rs~~, ~~acks~~	~~ph~~, ~~rs~~, acks	~~ph~~, acks	ph, ~~acks~~
Stable `rs`	~~ph~~, ~~rs~~, ~~acks~~	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks

Legend: ~~abc~~ = never happens, ~~abc~~ = erased, abc = kept forever

For a more complete explanation of this efficient presumed commit, see the paper by Lampson and Lomet.[1]

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 spec. As with the ordinary concurrency described in handout 20, there are many different specs 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 spec, 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. Our goal is to capture this constraint—that committed transactions are serializable in the order in which they commit—in a spec 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 `t1` has committed before it processes an operation for transaction `t2`, then `t2` must follow `t1` in the global commit order (assuming that `t2` 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 spec. (Rather than giving all the details, we sketch how to modify the spec of concurrent transactions given in handout 20.)

- Keep track of a partial order `precedes` on transactions, which should record that `t1 precedes t2` whenever the `Commit` procedure for `t1` happens before `Do` for `t2`. 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` on each `Do(B),` by adding all pairs `(tt, t2)` where `t1` is known to be committed.
- Change the constraint `Serializable` in the invariant in the spec to require serializability in all total orders consistent with `precedes`, rather that just some total order consistent with `xc`. Note 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.

[1] B. Lampson and D Lomet, A new presumed commit optimization for two phase commit. *Proc. 19th VLDB Conference*, Dublin, 1993, pp 630-640.