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,  
  S WITH { s0: () -> S }  
]  
EXPORT Begin, Do, Commit, Abort, Crash =  

TYPE A = S -> (V, S)  

VAR ss := S.s0()  
VAR vs := S.s0()  
VAR ph : ENUM[Idle, Run, Failing] := idle

% Value of an action  
% State  
% Action  
% Stable State  
% Volatile State  
% 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 >>  

PROC 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, 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 [ % Value of an action
  V, % State
  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 % Ph (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) =
  IF ph # idle => VAR v | (v, vs) := a(vs); RET v
  [ ] ph # run => RAISE crashed FI >>

APROC Prepare() RAISES (crashed) =
  IF ph = run => ps := vs; ph := idle [*] RAISE crashed >>

APROC Commit() = << ps ≠ nil => ss := ps; ps := nil [*] SKIP FI >>

APROC Abort () = << vs := ss, ph := idle [*] ps := nil >>

APROC Crash () =
  [ ] 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 directly through to the appropriate

CLASS Coord [ % Value of an action
  V, % State
  S WITH { s0: ()→S }, % State
  RM ]

EXPORT Begin, Do, Commit, Abort, Prepare, Crash =

TYPE R = RMTr % instance name on an RM
  Ph = ENUM[idle, commit] % Ph (volatile)

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 [ % instance name on an RM
  V, % State
  S WITH { s0: ()→S }, % State
  RM ]
CONST AtoRM : A -> RM := ... % the RM that handles action a
VAR ph rs finish : % 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 % the RM that handles action
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

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

PROC Do(a) -> V RAISES (crashed) = % abstractly a.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 := rs [*] SKIP FI;
  r.do(a)
  [*] RAISE crashed FI

PROC Commit() RAISES (crashed) = % assuming a partitioned state space, with S as a function from state component names to values; + combines these
  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 := {}

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

FUNC AtoR(a) -> (R + Null) = VAR r := AtoRM(a) |
  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 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-
sets \( \text{ph} \) 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 \text{Begin} on that RM. One way to handle this is for each client process to send \text{Begin} to the coordinator, which then calls \text{Begin} exactly once on each RM; this is what \text{Coord} does. This costs extra messages, however. An alternative is to eliminate \text{Begin} and instead have both \text{Do} and \text{Prepare} report to the client whether the transaction is new at that RM, that is, whether \( \text{ph} = \text{idle} \) before the action; this is equivalent to having the RM tell the client that it did a \text{Begin} along with the \text{Do} or \text{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):

\[
\begin{aligned}
\text{VAR} & \quad \text{news} = C \rightarrow R \rightarrow \text{Int} := \left\{ * \rightarrow 0 \right\} \\
\text{We add to the RM state a history variable } & \quad \\
\text{lost} \text{ which is true if the RM has failed and lost } & \quad\text{some of the client’s state. This is what the client needs to detect, so we maintain the invariant (here} \\
\text{clientPrs}(t) & \quad \text{is the set of client processes for } t:\} \\
\text{( ALL } r | r.t = t /\ r.\text{lost } \rightarrow & \quad \text{r.ph = idle } /\ r.\text{ps} = \text{nil}) \\
\text{\bigvee & \quad \{ + : \left\{ c : \text{IN clientPrs}(t) \mid \text{news}(c)(r) \right\} > 1 \right\} )
\end{aligned}
\]

After all the RMs have prepared, they all have \( r.\text{ps} \neq \text{nil} \), so if anything is lost is shows up in the \text{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 \text{Do} and \text{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.

\textit{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 \text{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 forget 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 \text{Prepare}, in which case \text{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; \text{Coord} does the former. Both look expensive. We can make remembering the outcome cheap, however, if we can compute it as a function \text{Presumed} of the transaction id \( t \) for any transaction that is pending.

The simplest way to use these ideas is to make \text{Presumed}(t) = \text{aborted} always, and to make \( rs \text{ stable at Commit} \), rather than at the first \text{Prepare}, by writing it into the commit record. This makes aborts cheap: it avoids an extra log write before \text{Prepare} and any acks for aborts. However, it still requires each RM to acknowledge the \text{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 \text{Presumed}(t) = \text{committed}. This is not straightforward, however, because now \text{Presumed} is not constant. Between the first \text{Prepare} and the commit, \text{Presumed}(t) = \text{aborted} because the outcome after a crash is \text{aborted} and there’s no stable record of \( t \), but once the transaction is committed the outcome is \text{committed}. This is no problem for \text{t.Commit}, which makes \( t \) explicit by setting \( \text{ph} := \text{commit} \) (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, \text{Presumed}(t) must have changed to \text{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, \text{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 \text{Presumed}(t) to \text{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 \text{Presumed} can change over time:

\[
\begin{array}{c|c|c|c}
\text{PC} & \text{PA-live} & \text{future} \\
\hline
\text{PresumeCommitted} & \text{PC} & \text{PA-live} & \text{future} \\
\hline
\text{Crash} & \text{PC} & \text{PA+\text{ph}=c} & \text{PA-live} & \text{future} \\
\hline
\text{PresumeCommitted} & \text{PC} & \text{PA+\text{ph}=c} & \text{PC} & \text{PA-live} & \text{future}
\end{array}
\]

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 \( rs \) \text{stable} as part of the \text{Commit}, which is cheap. We can avoid the permanent PA section entirely by making \( rs \) \text{stable before \text{Prepare}}, which is not cheap. The following table shows the various cost tradeoffs.
Commit, no crash  Commit, crash  Abort, no crash  Abort, crash

<table>
<thead>
<tr>
<th>Coord</th>
<th>Stable rs</th>
<th>ph, za, acks</th>
<th>ph, za, acks</th>
<th>ph, za, acks</th>
<th>ph, za, acks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presumed abort</td>
<td>Volatile</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
</tr>
<tr>
<td></td>
<td>Stable on commit</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
</tr>
<tr>
<td>Presumed commit</td>
<td>Volatile</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
</tr>
<tr>
<td></td>
<td>Stable on commit</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
<td>ph, za, za, acks</td>
</tr>
</tbody>
</table>

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 \( t_1 \) has committed before it processes an operation for transaction \( t_2 \), then \( t_2 \) must follow \( t_1 \) in the global commit order (assuming that \( t_2 \) 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 \( \text{precedes} \) on transactions, which should record that \( t_1 \) \( \text{precedes} \) \( t_2 \) whenever the Commit procedure for \( t_1 \) happens before Do for \( t_2 \). This can be done either by keeping a history variable with all external operations recorded (and defining