Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 27 April 30, 1997

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 'servers'. 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 implementations, and the big system can be built even though it wasn't thought of when the smaller ones were designed.

Specifications

We have to solve two problems: composing the separate servers 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 servers; two servers that implement the `SequentialTr` spec in handout 18 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 server 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 implementations we have in mind, the client may be invoking `Do` actions at several servers. 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 servers, since we don't want each server to have to verify that no other server 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 `failed` to the phase. A crash sets the phase to `failed`, which enables an internal `CrashAbort` action that aborts the transaction. In the meantime a `Do` can either succeed or raise `crashed`.

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

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

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

APROC Do(a) -> V RAISES {crashed} = <<
    IF   ph # idle => VAR v | (v, vs) := a.meaning(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 := failed >>

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

END DistSeqTr
```

In a real system `Begin` 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. If the concurrency spec is 'any can commit', `Do(t)` sees `vs = ss + actions(t)`, and `Commit(t)` does `ss := ss + actions(t)`.

Partial failures

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

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 servers requires a server action to make a transaction's state stable without committing it; this action is traditionally called `Prepare`. We can invoke `Prepare` on each server, and if they all succeed, we can commit the transaction. Without `Prepare` we might commit the transaction, only to learn that some server has failed and lost the transaction state.

The old `LogRecovery` or `LogAndCache` code in handout 18 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 `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 server 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 servers and telling them to commit or abort. Real systems sometimes allow the servers to query the coordinator, but we omit this minor variation.

We give the spec for a server. Since we want to be able to compose servers repeatedly, we give it as a modification of the `DistSeqTr` client spec. The change 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.

```
MODULE TrServer [
    V,                                          % Value of an action
    S WITH { s0: ()->S },                       % State
    A WITH { meaning: A->S->(V, S) }            % Action
    ] EXPORT Begin, Do, Commit, Abort, Prepare, Crash =

VAR ss          := S.s0()                       % Stable State
    ps          :  (S + Null) := nil            % Prepared State (stable)
    vs          := S.s0()                       % Volatile State
    ph          : ENUM[idle, run, failed] := idle   % PHase (volatile)

% INVARIANT ps # nil  ==>  ph = idle

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

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

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

APROC Commit() RAISES {crashed} = <<
    IF ps # nil => ss := ps; ps := nil [*] Abort(); RAISE crashed FI >>

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

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

END TrServer
```

This spec requires its client to call `Prepare` exactly once before `Commit`, and confusingly raises `crashed` in `Do` after `Prepare`. A real system might handle these variations somewhat differently, but the differences are inessential.

We don't give implementations of this spec, since they are very similar to `LogRecovery` or `LogAndCache`. Like the old `Commit`, `Prepare` forces the log to stable storage; then it writes a `prepared` record. `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.

Committing a transaction

We have not yet explained how to implement `DistSeqTr` using several copies of `TrServer`. The basic idea is simple. A coordinator keeps track of all the servers 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 servers, 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 servers to prepare. This succeeds if all the `Prepare`'s return normally. Then the coordinator records stably that the transaction committed, and tells all the servers to commit.

The abstraction function from the states of the coordinator and the servers to the state of `DistSeqTr` is simple. We need names for the servers:

```
TYPE R          =  Int                          % seRver name
```

The coordinator's state is

```
VAR ph          :  ENUM[idle, committed] := idle
    servers     :  SET R := {}
```

The server states are

```
VAR s           :  R -> [ss: S, ps: (S + Null), vs: S, ph]
```

The spec's `vs` is the union of all the server `vs` values. The spec's `ss` is the union of the servers' `ss` unless `ph = committed`, in which case any server with a non-`nil` `ps` substitutes that:

```
DistSeqTr.ss = + : {r :IN servers |
                    (ph = committed /\ s(r).ps # nil => s(r).ps [*] s(r).ss)}
```

We need to maintain the invariant that any server whose phase is not idle or which has `ps # nil` is in `servers`, so that it will hear from the coordinator what it should do.

If some server has failed, its `Prepare` will raise `crashed`. In this case the coordinator tells all the servers to abort, and raises `crashed` to the client. A server that is not prepared and doesn't hear from the coordinator can abort on its own. A server that is prepared cannot abort on its own, but must hear from the coordinator whether the transaction has committed or aborted.

This entire algorithm is called "two-phase commit"; do not confuse it with two-phase locking. The first phase is the prepares, the second the commits. The coordinator can use any algorithm it likes to record the commit or abort decision. However, once some server is prepared, losing this information will leave that server 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 servers 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 server 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 server does `Allow(abort)`, and when the coordinator has heard that every server 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 servers are not replicated, and one of the servers 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 servers 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 the coordinator's work. In particular, how does the coordinator keep track of the servers efficiently. This problem has three aspects.

Keeping track of servers

The first is simply finding out who the servers are, since the client may be spread out over many machines, and it isn't efficient to funnel every request to a server through the coordinator. The standard way to handle this is to arrange all the client processes in a tree, and require that each client process report to its parent the servers that it or its children have talked to. Then the root of the tree will know about all the servers, and it can either act as coordinator or give the coordinator this information.

Noticing failed servers

The second is noticing when a server has failed. 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 there may be lots of client processes, a client doesn't know the first time it talks to a server, so it doesn't know when to call `Begin` on that server. One way to handle this is for each client process to send `Begin` to the coordinator, which then calls `Begin` exactly once on each server. 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 server, that is, whether `ph = idle` before the action. If a server 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 server reports 'new' more than once, it must have crashed.

To make this precise, each client processes counts the number of 'new' reports it has gotten from each server (here `C` names the client processes):

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

We add to the server state a history variable `lost` which is true if the server has failed and lost some of the client's state. This is what the client needs to detect, so we maintain the invariant

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

After all the servers have prepared, they all have `s(r).ps # nil`, so if anything is lost is shows up in the `news` count.

A variation on this scheme has each server 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 server with more than one id that is prepared must have failed during the transaction.

Cleaning up

The third aspect of bookkeeping is making sure that all the servers find out whether the transaction committed or aborted. Actually, only the prepared servers need to find out, since a server that isn't prepared can just abort the transaction if it is left in the lurch.

The simple way to handle this is for the coordinator to record its `servers` variable stably before doing any prepares. Then even if it fails, it knows what servers to notify after recovery. However, this means an extra log write for `servers` before any prepares, in addition to the essential log write for the commit record.

You might try to avoid this write by just telling each server the identity of the coordinator, and having a server query for the transaction outcome. This doesn't work, because the coordinator needs to be able to forget the outcome eventually, in order to avoid the need to maintain state about each transaction forever. It can only forget after every server has learned the outcome and recorded it stably. If the coordinator doesn't know the set of servers, it can't know when all of them have learned the outcome.

If there's no stable record of the transaction, we can assume that it aborted. This convention is highly desirable, since otherwise we would have to do yet another log write at the beginning of the transaction. Given this, we can log the set of servers along with the commit record, since the transaction aborts if the coordinator fails before writing the commit record. But we still need to hear back from all the servers that they have recorded the transaction commit before we can clean up the commit record. If it aborts, we don't have to hear back, because of the convention that a transaction with no record must have aborted. This convention is called 'presumed abort'.

Since transactions usually commit, it's unfortunate that we have optimized for the abort case. To fix this, we can make a more complicated convention based on the values of transaction identifiers `T`. We impose a total ordering on them, and record a stable variable t_{low}. Then we maintain the invariant that any transaction with identifier $< t_{low}$ is either committed, or not prepared at any server, or stably recorded as aborted at the coordinator. Thus old transactions are 'presumed commit'. This means that we don't need to get acknowledgments from the servers for a committed transaction `t`. Instead, we can clean up their log entries as soon as $t < t_{low}$.

The price for this scheme is that we do need acknowledgements from the servers for aborted transactions. That is OK, since aborts are assumed to be rare. However, if the coordinator crashes before writing a commit record for `t`, it doesn't know who the servers are, so it doesn't know when they have all heard about the abort. This means that the coordinator must remember forever the transactions that are aborted by its crashes. However, there are not many of these, so the cost is small. 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 specification. As with the ordinary concurrency described in handout 19, 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. 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.)

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

- Keep track of a partial order `precedes` on transactions, which should record that *A* `precedes` *B* whenever the `Commit` procedure for *A* happens before `Do` 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` on each `Do(B),` by adding all pairs (*A*, *B*) where *A* is known to be committed.
- Change the constraint `Serializable` in the invariant in the specification 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.