

If the system uses version histories for atomicity, the hierarchy of Figure 9.36 can be directly implemented by linking outcome records. If the system uses logs, a separate table of pending transactions can contain the hierarchy, and inquiries about the state of a transaction would involve examining this table.

The concept of nesting transactions hierarchically is useful in its own right, but our particular interest in nesting is that it is the first of two building blocks for multiple-site transactions. To develop the second building block, we next explore what makes multiple-site transactions different from single-site transactions.

### 9.6.3 Multiple-Site Atomicity: Distributed Two-Phase Commit

If a transaction requires executing component transactions at several sites that are separated by a best-effort network, obtaining atomicity is more difficult because any of the messages used to coordinate the transactions of the various sites can be lost, delayed, or duplicated. In Chapter 4 we learned of a method, known as Remote Procedure Call (RPC) for performing an action at another site. In Chapter 7[on-line] we learned how to design protocols such as RPC with a persistent sender to ensure at-least-once execution and duplicate suppression to ensure at-most-once execution. Unfortunately, neither of these two assurances is exactly what is needed to ensure atomicity of a multiple-site transaction. However, by properly combining a two-phase commit protocol with persistent senders, duplicate suppression, and single-site transactions, we can create a correct multiple-site transaction. We assume that each site, on its own, is capable of implementing local transactions, using techniques such as version histories or logs and locks for all-or-nothing atomicity and before-or-after atomicity. Correctness of the multiple-site atomicity protocol will be achieved if all the sites commit or if all the sites abort; we will have failed if some sites commit their part of a multiple-site transaction while others abort their part of that same transaction.

Suppose the multiple-site transaction consists of a coordinator Alice requesting component transactions X, Y, and Z of worker sites Bob, Charles, and Dawn, respectively. The simple expedient of issuing three remote procedure calls certainly does not produce a transaction for Alice because Bob may do X while Charles may report that he cannot do Y. Conceptually, the coordinator would like to send three messages, to the three workers, like this one to Bob:

From: Alice  
To: Bob  
Re: my transaction 91

**if** (Charles does Y **and** Dawn does Z) **then do** X, please.

and let the three workers handle the details. We need some clue how Bob could accomplish this strange request.

The clue comes from recognizing that the coordinator has created a higher-layer transaction and each of the workers is to perform a transaction that is nested in the higher-layer transaction. Thus, what we need is a distributed version of the two-phase commit protocol. The complication is that the coordinator and workers cannot reliably

communicate. The problem thus reduces to constructing a reliable distributed version of the two-phase commit protocol. We can do that by applying persistent senders and duplicate suppression.

Phase one of the protocol starts with coordinator Alice creating a top-layer outcome record for the overall transaction. Then Alice begins persistently sending to Bob an RPC-like message:

From: Alice  
To: Bob  
Re: my transaction 271

Please do X as part of my transaction.

Similar messages go from Alice to Charles and Dawn, also referring to transaction 271, and requesting that they do Y and Z, respectively. As with an ordinary remote procedure call, if Alice doesn't receive a response from one or more of the workers in a reasonable time she resends the message to the non-responding workers as many times as necessary to elicit a response.

A worker site, upon receiving a request of this form, checks for duplicates and then creates a transaction of its own, but it makes the transaction a *nested* one, with its superior being Alice's original transaction. It then goes about doing the pre-commit part of the requested action, reporting back to Alice that this much has gone well:

From: Bob  
To: Alice  
Re: your transaction 271

My part X is ready to commit.

Alice, upon collecting a complete set of such responses then moves to the two-phase commit part of the transaction, by sending messages to each of Bob, Charles, and Dawn saying, e.g.:

Two-phase-commit message #1:

From: Alice  
To: Bob  
Re: my transaction 271

PREPARE to commit X.

Bob, upon receiving this message, commits—but only tentatively—or aborts. Having created durable tentative versions (or logged to journal storage its planned updates) and having recorded an outcome record saying that it is `PREPARED` either to commit or abort, Bob then persistently sends a response to Alice reporting his state:

### Two-phase-commit message #2:

From: Bob  
To: Alice  
Re: your transaction 271

I am PREPARED to commit my part. Have you decided to commit yet? Regards.

or alternatively, a message reporting it has aborted. If Bob receives a duplicate request from Alice, his persistent sender sends back a duplicate of the PREPARED or ABORTED response.

At this point Bob, being in the PREPARED state, is out on a limb. Just as in a local hierarchical nesting, Bob must be able either to run to the end or to abort, to maintain that state of preparation indefinitely, and wait for someone else (Alice) to say which. In addition, the coordinator may independently crash or lose communication contact, increasing Bob's uncertainty. If the coordinator goes down, all of the workers must wait until it recovers; in this protocol, the coordinator is a single point of failure.

As coordinator, Alice collects the response messages from her several workers (perhaps re-requesting PREPARED responses several times from some worker sites). If all workers send PREPARED messages, phase one of the two-phase commit is complete. If any worker responds with an abort message, or doesn't respond at all, Alice has the usual choice of aborting the entire transaction or perhaps trying a different worker site to carry out that component transaction. Phase two begins when Alice commits the entire transaction by marking her own outcome record COMMITTED.

Once the higher-layer outcome record is marked as COMMITTED or ABORTED, Alice sends a completion message back to each of Bob, Charles, and Dawn:

### Two-phase-commit message #3

From: Alice  
To: Bob  
Re: my transaction 271

My transaction committed. Thanks for your help.

Each worker site, upon receiving such a message, changes its state from PREPARED to COMMITTED, performs any needed post-commit actions, and exits. Meanwhile, Alice can go about other business, with one important requirement for the future: she must remember, reliably and for an indefinite time, the outcome of this transaction. The reason is that one or more of her completion messages may have been lost. Any worker sites that are in the PREPARED state are awaiting the completion message to tell them which way to go. If a completion message does not arrive in a reasonable period of time, the persistent sender at the worker site will resend its PREPARED message. Whenever Alice receives a duplicate PREPARED message, she simply sends back the current state of the outcome record for the named transaction.

If a worker site that uses logs and locks crashes, the recovery procedure at that site has to take three extra steps. First, it must classify any PREPARED transaction as a tentative winner that it should restore to the PREPARED state. Second, if the worker is using locks for

before-or-after atomicity, the recovery procedure must reacquire any locks the PREPARED transaction was holding at the time of the failure. Finally, the recovery procedure must restart the persistent sender, to learn the current status of the higher-layer transaction. If the worker site uses version histories, only the last step, restarting the persistent sender, is required.

Since the workers act as persistent senders of their PREPARED messages, Alice can be confident that every worker will eventually learn that her transaction committed. But since the persistent senders of the workers are independent, Alice has no way of ensuring that they will act simultaneously. Instead, Alice can only be certain of eventual completion of her transaction. This distinction between simultaneous action and eventual action is critically important, as will soon be seen.

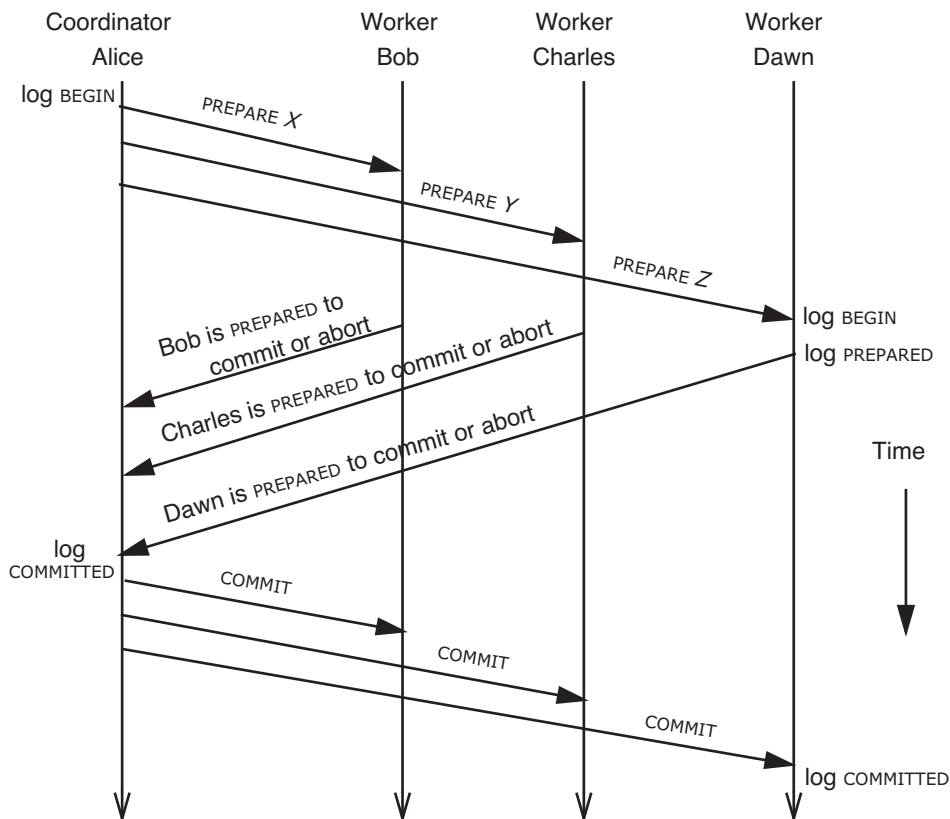
If all goes well, two-phase commit of  $N$  worker sites will be accomplished in  $3N$  messages, as shown in Figure 9.37: for each worker site a PREPARE message, a PREPARED message in response, and a COMMIT message. This  $3N$  message protocol is complete and sufficient, although there are several variations one can propose.

An example of a simplifying variation is that the initial RPC request and response could also carry the PREPARE and PREPARED messages, respectively. However, once a worker sends a PREPARED message, it loses the ability to unilaterally abort, and it must remain on the knife edge awaiting instructions from the coordinator. To minimize this wait, it is usually preferable to delay the PREPARE/PREPARED message pair until the coordinator knows that the other workers seem to be in a position to do their parts.

Some versions of the distributed two-phase commit protocol have a fourth acknowledgment message from the worker sites to the coordinator. The intent is to collect a complete set of acknowledgment messages—the coordinator persistently sends completion messages until every site acknowledges. Once all acknowledgments are in, the coordinator can then safely discard its outcome record, since every worker site is known to have gotten the word.

A system that is concerned both about outcome record storage space and the cost of extra messages can use a further refinement, called *presumed commit*. Since one would expect that most transactions commit, we can use a slightly odd but very space-efficient representation for the value COMMITTED of an outcome record: non-existence. The coordinator answers any inquiry about a non-existent outcome record by sending a COMMITTED response. If the coordinator uses this representation, it commits by destroying the outcome record, so a fourth acknowledgment message from every worker is unnecessary. In return for this apparent magic reduction in both message count and space, we notice that outcome records for aborted transactions can not easily be discarded because if an inquiry arrives after discarding, the inquiry will receive the response COMMITTED. The coordinator can, however, persistently ask for acknowledgment of aborted transactions, and discard the outcome record after all these acknowledgments are in. This protocol that leads to discarding an outcome record is identical to the protocol described in Chapter 7[on-line] to close a stream and discard the record of that stream.

Distributed two-phase commit does not solve all multiple-site atomicity problems. For example, if the coordinator site (in this case, Alice) is aboard a ship that sinks after



**FIGURE 9.37**

Timing diagram for distributed two-phase commit, using  $3N$  messages. (The initial RPC request and response messages are not shown.) Each of the four participants maintains its own version history or recovery log. The diagram shows log entries made by the coordinator and by one of the workers.

sending the *PREPARE* message but before sending the *COMMIT* or *ABORT* message the worker sites are left in the *PREPARED* state with no way to proceed. Even without that concern, Alice and her co-workers are standing uncomfortably close to a multiple-site atomicity problem that, at least in principle, can *not* be solved. The only thing that rescues them is our observation that the several workers will do their parts eventually, not necessarily simultaneously. If she had required simultaneous action, Alice would have been in trouble.

The unsolvable problem is known as the *dilemma of the two generals*.