Peter Thanisch
Accrue Software

Commit protocols have been proposed for use in a variety of concurrent-computing applications. The author has developed two condition sets that can help you determine when to use a commit protocol and when to avoid them.

Atomic Commit in Concurrent Computing

Commit protocols were originally developed for distributed transaction processing (TP) to help provide atomicity and isolation for a set of actions. More recently, researchers and practitioners have proposed the use of commit protocols for many concurrent-computing applications, including atomic broadcast, distributed file systems, Internet TP, network management, and reliable messaging.

Designing a commit protocol for a particular application is tricky, especially when a distributed computing environment is unreliable and thus requires efficiency and robustness. Furthermore, developments in middleware[1] have made it relatively easy to incorporate industry-standard commit protocols into concurrent-computing applications.[2] However, potential users must bear in mind that these standard commit protocols were designed and optimized with a typical TP workload in mind.[3] According to industry folklore, most transactions are read-only, and the average transaction

- accesses 10 data items,
- executes no more than a million machine instructions,
- stands a 95% chance of succeeding (rather than aborting or being aborted), and
- spends one third of its lifecycle in the commit phase.

Because TP bears little resemblance to new application areas, transplanting a standard commit protocol into a concurrent computing application can take a toll on performance. The issue is further complicated by the fact that some computational problems appear to require a commit protocol but are actually either too easy or too hard to benefit from its use.[4]

To contend with these issues, I have outlined two sets of conditions that can help you determine whether or not to use a commit protocol for a concurrent-computing application. Here, I offer an overview of commit protocols. I then discuss atomicity and isolation in application-neutral terms and present my condition sets for atomicity and barrier synchronization.These condition sets can help you decide whether or not you need a commit protocol, and can also help you reengineer your application to avoid the need for one should it prove too expensive to implement.

Commit protocol overview

In transaction processing, distributed transactions read and update data items located on two or more autonomous database servers.[2] The commit protocol kicks in at the end of each transaction to help provide the "A" and "I" of the ACID prop-

1092-3063/00/$10.00 © 2000 IEEE

erties—Atomicity, Consistency, Isolation, and Durability.

Two or more related actions that change the outside world are *atomic* if the computation's correctness depends on adherence to the following rule:

> If circumstances external to the computation conspire to prevent any of the transaction's updates from being carried out, then none of the transaction's updates are carried out.[2]

Atomicity is typically required, for example, when you book a flight and hotel for a vacation. If all the flights are already full, your hotel booking is useless and vice versa. You need both, or, failing that, neither.

The commit protocol helps provide *isolation* when a transaction reads data items. When concurrently executing transactions continually update a database, the commit protocol creates a barrier synchronization that ensures that the transaction sees a globally consistent state of the distributed database. In a bank, for example, one transaction might total the balances in two accounts while another transaction transfers an amount from one of these accounts to the other. Isolation prevents the totaling transaction from reading one account before the transfer and the other account after the transfer.

In a distributed database system, the lock managers work in conjunction with the commit protocol to provide isolation. With respect to isolation, the commit protocol provides only barrier synchronization. Other aspects of the isolation service are provided by other database system components, such as the lock manager, and I do not discuss them here.

Protocol elements

Although implementation details vary depending on the application, I have identified three basic elements common to all commit protocols.

- *Context* is the computation from which the set of client requests emanates. The context represents a logical locus of control that can be single-threaded, multithreaded, multiprocess, or distributed.
- *Agents* are responsible for interactions with the outside world. Agents need not be physically distributed—they could be processes executing on different machines, separate processes executing on the same machine, or separate threads running in the same process. *Retrieving agents* obtain information about the state of the outside world. *Altering agents* change the state of the outside world. In many applications, a single agent performs both of these roles.
- A *coordinator* enlists the service of agents to perform atomicity and barrier synchronization, and thereafter initiates the commit protocol. Coordinator actions are based on instructions from the context.

A *decentralized* commit protocol avoids the need for a coordinator by having agents swap information with each other. However, the computational expense of such extra message passing is unacceptable in most applications. I therefore assume a *centralized* protocol in which the coordinator and context are separate (in some implementations, the coordinator is an OS service). I refer to the combined activity of all three elements as the *computation*.

Feasible and infeasible interactions

The new applications for commit protocols have at least one thing in common: the computation attempts to interrogate or alter the state of the outside world. Whenever an agent interacts with the outside world, its actions are potentially infeasible. As Table 1 shows, such infeasibility can arise for various reasons.

Table 1. Infeasibility and agent actions.

Type of agent	Interaction with the outside world	Possible cause of infeasibility
Retrieval	Reading a data item in a database.	The data item is exclusively locked by an agent in another concurrently executing computation.
Retrieval	Interrogating a device, such as a thermometer.	The device is inaccessible because of a physical fault.
Update	Writing a data item to a database.	The resulting state of the database would violate an integrity constraint.
Update	Creating a file in the operating system's file subsystem.	The disk is full.
Update	Activating a device, such as an alarm.	The computation is unauthorized.

The common factor in infeasible actions is that both the problem and the cure are outside the computation's control. Neither the context nor the agent can do much more than report the condition or, in the case of a transient fault, wait. In computations that use a commit protocol to help achieve atomicity, the agent must therefore check a proposed action's feasibility. For the commit protocol to be useful, it must be possible (at least in principle) for the agents to disagree with each other about feasibility. This implies that each agent must decide on the feasibility of its request *independent* of other agents. If all agents were always unanimous (and thus, within each separate computation, either all actions are feasible or all are unfeasible), then the coordinator would only need to check with one of the agents.

In some applications, an agent performs both retrievals and alterations. In typical TP applications, the context gathers information from the agents and uses that information to decide which alterations the agent should make. Thus, the context might send several requests to a particular agent and, taken collectively, they might be infeasible even though they are individually feasible. For example, their net effect might break an integrity constraint.

Let's examine how commit protocols provide atomicity and barrier synchronization.

A commit protocol for atomicity

A two-phase commit protocol for atomicity entails several steps.

1. Once the context completes computation and wants the agents to make their changes permanent, the context informs the coordinator to initiate the commit protocol.
2. The coordinator sends a `determine-feasibility` message to its enlisted agents and waits for them to reply.
3. When an agent receives the `determine-feasibility` message from the coordinator, it decides whether its subtask is feasible. If so, it sends the coordinator a `feasible` message; otherwise, it sends an `infeasible` message and aborts the subtask (if it has not already aborted it). If no message has arrived from the coordinator when a timeout expires, the agent unilaterally aborts its subtask and releases its computation-related resources.
4. Once the coordinator collects all agent replies, it makes one of three decisions:
   - If all agents send feasible messages, it decides to commit.
   - If one or more agents send an `infeasible` message, it decides to abort.
   - If one or more agents fail to send a response message after the timeout period expires, the coordinator decides to abort.
5. The coordinator then informs the context of the outcome and forwards the decision to agents that sent feasible messages. It need not reply to agents that sent infeasible messages.
6. When the agents receive the coordinator's message, they act accordingly.

Commit protocols for atomicity

An application requires atomicity whenever it uses two or more agents, at least one of which is an altering agent. The altering agent first checks the proposed change's feasibility. If a proposed change is infeasible, the agent can immediately release its context resources and the result of the commit protocol will be to abort the attempt to make a change. When the change is feasible, the next step depends on the system design.

- In an *optimistic system*, the agent makes externally visible changes; if the agent's context subsequently aborts, the system must abort all contexts that optimistically retrieved the agent-altered data.
- In a *conservative system*, the agent's changes are not made externally visible until the feasibility of the context's entire set of required retrievals and alterations is established.

The sidebar, "A commit protocol for atomicity," shows a basic version of the two-phase commit (2PC) protocol.[2]

Ensuring atomicity

In some applications, agents can undo the changes they make to the outside world. For example, the context might execute a *compensating* transaction to cancel out the first transaction's effects. Thus, the entire computation's atomicity is not permanently compromised if one agent performs its subtask and, subsequently, another agent announces that it can't perform its subtask. This simplifies the task of ensuring atomicity and eliminates the need for a commit protocol—unless the cost of undoing the agent's change is prohibitively expensive.

In some cases, it is impossible to undo a change's external effects. For example, once an automatic teller machine dispenses cash, the system cannot undo the action. In such cases, the subtask effects are *permanent*—the change's effects are visible and thus can influence behavior external to the computation. In practice, the computation can subsequently change part of the outside world's state. Nonetheless, the changes are still permanent in that the new values could have influenced behavior external to the computation.

Wherever the atomicity property is required and interactions with the outside world are potentially infeasible, the computation must proceed tentatively. It would be wrong to simply tell agents to perform their subtasks, because if one agent reports that its subtask is infeasible, the atomicity property is lost. Consequently, the coordinator must first consult each agent on its subtask's feasibility.

Although agents can check whether subtasks are feasible, in many applications they cannot prevent a change of external circumstances that might render the subtask infeasible. Without this complication, a simpler protocol would be possible because the context could deal with agents one by one. If controlling feasibility is impossible, atomicity cannot be ensured. A commit protocol is thus useful only in cases where a catastrophic failure, such as a disk head crash, would prevent the successful execution of an agent's feasible action.

Atomicity conditions

For a commit protocol to ensure atomicity when agent tasks change the outside world, an application should meet several conditions. To facilitate discussion and mapping to particular applications, I have assigned each condition a letter based on its key characteristic.

- (M) A computation must have *multiple agents* (at least one of which is an altering agent).
- (T) Agents *tentatively* establish feasibility, given that external circumstances can render the changes infeasible.
- (B) *Before* the agent votes (B1) external circumstances can render the changes infeasible at any time, and (B2) the context or coordinator can force the agent to abort at any time.
- (A) *After* the agent votes feasible (A1) only a catastrophic failure can render the changes infeasible, and (A2) the coordinator alone determines whether the agent commits or aborts.
- (P) Agent changes are *permanent* in that the agent can't undo them.

Whereas M is a property of the agents set, P and T are properties of how agents interact with the outside world. M and B1 ensure that the problem is sufficiently hard to warrant the use of a commit protocol, whereas T, B2, A1, A2, and P ensure that the problem is sufficiently easy. If your application meets *all* of the MTBAP conditions, it requires some-

thing approximating a commit protocol. Note, however, that these MTBAP properties are minimal. In any particular application, the commit protocol will face added demands and complications.

Performance issues

In many applications, some agents might fail or be cut off due to a partial network failure, while other agents continue communicating and carrying out their subtasks. Commit protocols are designed to survive various categories of partial failures. For example, the atomicity protocol shown in the sidebar "A Commit Protocol for Atomicity" uses timeouts to detect possible failures. For example, if an agent fails before voting, the coordinator will timeout and might decide to abort. The only effect on overall system performance is a short delay and the extra work associated with rolling back and restarting the context. However, other partial failures can cause the protocol to *block*, as the following example shows.

> Example 1. Suppose that there are three agents: A1, A2, and A3, and a coordinator, C. A1 and A2 have both sent `feasible` messages to C, but neither has received a subsequent message from C. A1 and A2 try to contact C in case a message has been lost or delayed, but discover that C and A3 have crashed or become isolated. A1 and A2 are thus blocked:
>
> - A1 and A2 cannot commit because, before it crashed, C might have told A3 to abort; and
> - A1 and A2 cannot abort because, before it crashed, C might have told A3 to commit and A3 might have carried out this command before it crashed.
>
> Given this, A1 and A2 must continue to hold locks until C or A3 recovers.

A partial failure that leads to blocking impacts overall system performance because the "live" agents continue to hold resources for the stalled client computation. Throughput of concurrently executing computations thus drops while they await the release of shared resources by agents stalled in the commit phase.

For a commit protocol to be robust with respect to a particular failure scenario, it must facilitate recovery regardless of how infrequently a particular failure scenario occurs. Traditionally, this involves logging to disk the information about a protocol execution's progress and outcome. Upon recovery from a system crash, the participant reads the log file and takes appropriate action to rejoin the protocol execution.

Commit phases

The industry standard (de jure and de facto) commit protocols[2] are 2PC protocols. In phase one, the participants establish whether or not there is unanimous feasibility. In the second phase, agents commit as soon as they are told that feasibility is global. However, as Example 1 shows, 2PC protocols can cause problems under various failure conditions.

Three-phase commit (3PC) protocols can help in such situations. In quorum-based 3PC protocols—where a quorum is a majority of the agents—the coordinator informs all agents when feasibility is unanimous. The agents then acknowledge receipt of this information, but do not, at this stage, commit. When the coordinator has collected acknowledgements from a quorum, it tells agents to commit. If the failure scenario outlined in Example 1 occurs, A1 and A2 initiate a *termination protocol*, as Example 2 shows.

> Example 2. If neither A1 nor A2 have been informed about the unanimity of feasibility, they realize that they constitute a quorum and thus that C cannot have received quorum acknowledgement. Hence, C could not have told A3 to commit and it is safe for A1 and A2 to abort. When C and A3 recover, they rejoin A1 and A2 in the commit-protocol execution and are informed of the outcome.

To date, 3PC has not been used in commercial TP systems. There are two primary reasons for this. First, 3PC involves extra messages for all transactions, regardless of the rarity of relevant failure scenarios. Second, some failure scenarios can cause 3PC to block.

At the other end of the scale, one-phase commit is sufficient for some applications. For example, in some applications, agents can immediately determine and report feasibility. Thus, by the time the commit-protocol execution starts, the context knows its global feasibility and the coordinator must merely inform the agents of the global outcome. Several services in new applications (described later) fit this category.

Commit protocols for barrier synchronization

For some distributed applications, atomicity alone is insufficient because it cannot guarantee that retrievals or alterations will happen simultaneously at all sites. Agents belonging to separate, concurrently executing computations can alter parts of the global state that the target computation needs to retrieve or alter. This is particularly problematic in applications such as distributed TP, which includes the notion of a *consistent global state* from which agents at various sites retrieve information. The agents' views must be compatible with each other. Atomicity alone cannot ensure this compatibility because a computation's alterations and retrievals at different sites can occur at different times, and thus other computations can make interim changes.

Industry-standard commit protocols[2] therefore provide isolation by enforcing *barrier synchronization* (see the sidebar, "A commit protocol for barrier synchronization"). In distributed computing environments, it is generally impossible to control retrieval such that agents will obtain their information at exactly the same time. The global state can change at any time, and although an agent might be able to *delay* such changes—by locking or versioning, for example—it cannot *veto* changes. In TP, for example, the system can break a deadlock by overriding an agent's attempts to retain a lock on a data item.

There are four conditions for best use of a commit protocol to ensure barrier synchronization for retrieving agents:

A commit protocol for barrier synchronization

A simple commit protocol for barrier synchronization involves four basic steps.

1. As the context proceeds with the computation, it asks the coordinator to enlist agents.
2. The context then issues requests to agents for retrievals and alterations; once these are complete, it tells the coordinator to initiate the commit protocol.
3. The coordinator asks the agents to confirm that they are still locking the state on behalf of the computation and tells retrieval agents to release their resources for other agents to use.
4. If all agents confirm that they are holding their locks, the coordinator tells the context that barrier synchronization was achieved. Otherwise, the coordinator tells the context that barrier synchronization might not have been achieved. The coordinator also informs the altering agents about the outcome, so that they can determine whether or not to make their changes permanent.

- (M) A computation features *multiple* independent agents.
- (A) Access time is *asynchronous*. There is no global time and the context has no direct control over the precise moment at which an agent reads the outside world.
- (F) A *fluid* global state prevents agents from freezing a part of the state to obtain a consistent snapshot.
- (G) Agents require a *globally* consistent state to see snapshots of the world that are consistent with each other.

If your application lacks one of these conditions, you can use a simpler protocol to ensure barrier synchronization. For example, if F does not hold, the coordinator can simply tell agents when to release their locks, simplifying steps 3 and 4 in the barrier synchronization protocol.

New applications

Researchers have developed commit protocols for several concurrent-computing applications. The demands of these applications differ in several ways from those in traditional TP.

First, traditional TP contains the notion of a consistent global state, in which each transaction transforms the database from one consistent global state to another. Indeed, one of the roles of the commit protocol is to ensure that agents only make changes that maintain this local consistency. However, some of the new applications contain nothing analogous to this notion of global consistency beyond the computation wanting atomicity to hold. For example, a context might want one agent to flash a light and another to briefly sound a buzzer. Although the context wants to enforce the atomicity of these two transient events, there is no real sense in which the system achieves a consistent state.

Second, in traditional TP, agents hold locks on shared resources until the commit protocol reaches a critical point. Researchers have optimized some commit protocols to permit early release of such locks. However, not all of the new applications need to lock critical data resources, as they don't compete for such resources with other concurrently executing computations. Consequently, several commit-protocol optimizations are irrelevant. Optimizations should focus on reducing an individual computation's response time. Boosting throughput by enhancing concurrency might not be significant.

Third, although the commit outcome is always preferred to the abort outcome, the penalty of having to abort differs greatly among applications. For example, in traditional TP, restarting a transaction is generally easy and, in many applications, doing so wastes little work. In contrast, some systems have jobs that take days to complete and restarting would be a major waste of resources. Given this, different applications require different protection against aborts.

Finally, some applications have real-time constraints on task completion. For example, each transaction might have a deadline for completion and be forced to abort if it misses that deadline. Some researchers have proposed special-purpose commit optimizations for real-time applications.

I now offer an overview of new applications and how they incorporate commit protocols. Note, however, that many algorithms exist for these applications and each application has alternative implementations that do not use commit protocols. Table 2 summarizes the application characteristics as they relate to the condition sets for atomicity and isolation.

Atomic broadcast

In some distributed computations, the nodes can broadcast messages to each other through a message delivery system. This type of computation requires two properties:

- *Atomicity*. If it is impossible for all nodes to receive a particular broadcast message, then none should receive it.
- *Uniform delivery sequence*. The system can deliver the stream of broadcast messages to application processes in any order, but the order must be the same for all processes.

Shyh-Wei Luan and Virgil Gligor use a quorum-based 3PC protocol in their atomic broadcast protocol.[5] At each node, an agent exchanges messages with agents at other nodes. The agent also delivers the broadcast messages it has received to its local application process. When an agent receives broadcast messages and wants to deliver them to its application process, it must ensure that another agent has not already delivered messages to its application process in a different order. The agent's node becomes an *initiator*, which combines the context and coordinator roles. The initiator polls agents at other nodes to see if they can deliver the proposed message sequence. The initiator agent also sends out details of the message sequence it has already delivered.

An agent will send back a vote of `infeasible` if it has already delivered one or more messages to its application process that the initiator's agent has yet to deliver. Included in the `infeasible` message will be the missing message sequence. If the initiator's agent receives an `infeasible` message, it aborts the

Table 2. New applications of commit protocols as they relate to condition criteria.

Service	Phases	Atomicity							Barrier synchronization			Extra difficulty	Outcome qualified by vote?
		M	T	B1	B2	A1	A2	P	M	A	F		
Atomic broadcast	3	y	n	n	y	y	y	y	y	y	n	n	y
Dist. file system	1	y	y	n	y	n	y	y	y	y	n	n	n
Dist. shared memory	1 or 2	y	n	n	y	y	y	n	y	y	n	n	y
Internet TP	2	y	y	y	y	y	y	y	y	y	y	y	n
Network mgt.	2	y	y	n	y	y	y	n	y	n	y	n	y
Reliable messaging	2	y	y	n	y	y	y	y	-	-	-	n	n
Remote backup	2	y	y	n	y	y	y	y	-	-	-	y	n
Replication	2 or 3	y	y	y	y	y	y	y	-	-	-	n	n
Scheduling	2	y	y	n	y	y	y	n	y	n	n	n	n

protocol execution. Because the agent did not participate in an earlier, successful protocol execution, it must catch up by delivering the missing messages. If an agent has not delivered a longer sequence of messages than the initiator's agent, it votes `feasible` and requests any missing message from the initiator's agent so that it can catch up.

Because the agents are not truly independent of each other, the initiator only needs a quorum of `feasible` votes. Once it has a quorum, it sends the details of the addition to the agents' streams, along with a list of agents that sent `feasible` responses. Agents send `Acks` upon receipt. When the initiator receives a quorum of `Acks`, it instructs the quorum nodes to commit the new message by adding it to their streams.

Luan and Gligor's protocol also includes

- a mechanism for dealing with concurrent protocol executions with different initiators, and
- a termination protocol that kicks in when a node suspects that the initiator has failed.

Table 3 shows the key differences between this broadcast application and distributed TP.

Distributed file systems

Rajmohan Panadiwal and Andrzej Goscinski[6] developed a high-performance distributed file-storage system that uses a commit protocol when writing changed pages back to their file. As the commit protocol proceeds, it determines an efficient technique for making changes permanent, such as write-ahead logging for contiguous pages and shadow paging for noncontiguous pages. This preliminary information gathering lets the commit protocol proceed in a single phase.

Distributed shared memory

Jingsong Ouyang and Gernot Heiser[7] provide fault tolerance in distributed shared-memory applications by using 2PC in a consistent checkpointing technique. The coordinator makes tentative checkpoints after the first phase of 2PC. In the second phase, the coordinator can use a lazy approach by piggybacking the results sent to agents on the messages associated with the next checkpoint. Ouyang and Heiser point out that a one-phase commit is possible, but agents must store the *two* most recent checkpoints to ensure recovery, whereas with 2PC, they need only store the most recent one. This technique also makes the simplifying assumption that only one distributed process is performing a checkpoint at any one time. This eliminates the F in MAFG.

Internet TP

Internet transactions need a commit protocol in the same way that conventional transactions do. However, for most concurrent applications, we can assume that the participating agents are cooperative. This assumption might be invalid in applications such as e-commerce,[8] where agents participating in an Internet transaction can represent different enterprises. In such cases, a participating enterprise might try to influence the transaction's outcome to its advantage. Consequently, a robust commit protocol for such an environment must have some of the characteristics of a harder problem—the Byzantine agreement,[4] which handles cases in which bogus messages are sent to deliberately fool the agents or the coordinator.

Table 3. Differences between a conventional commit protocol and Luan and Gligor's commit protocol.

Transaction processing	Atomic broadcast
A scheduler decides the order of transactions before initiating the commit protocol.	The commit protocol decides the ordering of message delivery.
An agent votes `infeasible` when there is a problem (external to the protocol) at its own site.	An agent votes `infeasible` if the *initiator* has a problem (that is, it hasn't received all of the broadcasts that it should have).
Contention for resources among transactions is a major performance factor.	There is no contention for shared resources among concurrently executing initiators.
The coordinator must receive a vote from every received agent before it can commit.	The initiator decides commit once it has feasible votes from a quorum of agents.
Both barrier synchronization and atomicity are required.	Atomicity is the only service required.

There is no notion of a consistent global state because the various participating systems belong to separate enterprises. Hence atomicity is the only service required.

Network management

C.S. Li and colleagues propose a commit protocol for network management to facilitate connection setup in switching systems that support mixed-circuit and packet connections for broadband service.[9] Their control architecture is intended for systems that allow only one connection to an output port at any given time.

Various users might demand different connection styles, such as broadcast, multicast, and conferencing. Given this, the developers note that a centralized system is preferable for some services, while for others a distributed arrangement might be more efficient. Their proposed scheme is an attempt to get the best of both worlds.

The developers describe their protocol as having three phases, but I refer to it as a two-phase protocol because one phase involves no message passing. The developers assume that their phases are synchronized, as are incoming requests. This departs from traditional assumptions made in the literature about commit protocols, but is not unreasonable. In traditional commit-protocol processing, the protocol uses the network, which might be congested. In this application, the network uses the commit protocol. A consequence of the developers' synchronization assumptions is that the protocol can be decentralized and does not require a separate coordinator. Furthermore, the assumption obviates the need for barrier synchronization.

The commit protocol keeps information on the port-allocation status consistent in the various controllers. Li and colleagues propose a multilevel hierarchical architecture in which virtual-centralized controllers (VCCs) are attached to each level of the control network. A VCC can obtain a port's current status from its VCC. Apart from the top and bottom of this hierarchy, each VCC acts as a combined agent, context, and coordinator. A VCC is an agent in the commit protocol that is initiated by a VCC higher up the hierarchy; it in turn acts as context and coordinator for the VCCs lower in the hierarchy. The protocol actually comprises many concurrent but synchronized commit protocols. Its three phases are

- Phase 1: Connection request processing. Each VCC, VCC_i, makes a tentative decision on whether it can grant the connection requests, based on the port status at the previous cycle's conclusion. A tentative allocation, T_I, is made by VCC_i.
- Phase 2: Tentative allocation broadcast. Tentative decisions are sent up and down the hierarchy. Thus, each VCC collects the tentative decisions made by its parents and children.
- Phase 3: Conflict resolution. Conflicts that occurred in the tentative allocations are resolved and a final allocation is generated. Various conflict-resolution rules are possible.

Finally, none of the ports on a destination list of a multicast or a broadcast connection will be allocated unless *all* of them are available.

Reliable messaging

Message queue mangers provide reliable asynchronous communication between processes in a distributed system.[10] However, they provide more services than simply sending a message to a single recipient. They can provide atomicity, ensuring that a message gets to a particular set of sites, rather than to just some of them. Each receiving site has a local message queue manager, and the sending site ensures that the message is placed in reliable queues so that all sites can receive it.

The process of receiving messages is largely nontransactional: only specialized applications require barrier synchronization. Furthermore, if processing a particular message causes an error, rolling back and putting the message back into the queue is pointless if the same error will occur the next time the message is read. To deal with this, application-specific error handling is required.

The relationship between reliable messaging and TP is convoluted. The leading reliable messaging products use industry-standard commit protocols; another product uses a database system to ensure message persistence.

Remote backup

Sharad Mehrotra and colleagues describe a remote backup technique for a multicomputer system.[11] The technique replicates updates onto a backup system. However, because of the application's nature, it makes no sense to use a conventional commit protocol. The point of doing this type of backup is to survive a failure—if the processor that fails happens to house the coordinator, there is still a single point of failure. The developers' solution is to duplicate the coordinator process on separate nodes. At commit time, copies are made to backup computers, which confirm that the copies are on stable storage to the backup coordinator. The backup coordinator then acts as a participant in the commit with the primary computers.

Replication

Some data items are replicated to only a few sites. Whenever a transaction alters such a data item, all of the sites it is replicated on can become commit-protocol agents to ensure the altering propagation's atomicity. However, this strategy is impractical for data replicated on numerous sites, which has led some researchers to explore ways of relaxing the atomicity requirement.

Liu and colleagues propose a scheme whereby the sites that replicated data should be propagated to are organized into a tree-structured hierarchy.[12] At each hierarchy level, a transaction with 2PC propagates the replicated data item to the sites at the next level down. The propagation transaction is executed and committed independent of the original transaction. The drawback is that at any time a data item might have different values on different sites.

Ididt Keidar and Danny Dolev[13] propose a quorum-based 3PC protocol,

where a quorum would be a majority of sites on which a data item is replicated. Their protocol can survive partial failures without blocking as long as a quorum of the replicating sites continues communicating.

SCHEDULING

Paolo Bizzarri[14] and colleagues developed a multicomputer scheduling technique in which each system node can potentially carry out its computation using several different computational procedures. They use a 2PC protocol in which the agents (the nodes) vote on feasibility and, at the same time, inform the coordinator of the different procedures. On receiving these replies, the coordinator not only determines feasibility, but also formulates a compatible plan for carrying out the computation.

RESEARCHERS ORIGINALLY DEVISED commit protocols to facilitate distributed transactions between multiple mainframes. However, many enterprises are now replacing monolithic software systems with a complex infrastructure consisting of autonomous midtier components and back-tier server applications. These components and server applications might be supplied by different vendors and communicate with each other through some form of message passing. Researchers are now deploying commit protocols to coordinate actions involving shared resources in multitier client–server systems with distributed objects middleware.[1] However, unlike traditional TP, the agents are not necessarily complex or robust industrial-strength systems. In fact, theoretically, anyone could write an agent to join a commit-protocol execution.

This is one among many challenges that commit-protocol designers face as they develop these protocols for areas such as mobile computing, the Internet, and client–server computing. In the past, for example, people typically developed commit protocols for tightly coupled computational environments where message latency was not only predictable but was less significant than disk latency. In some of the new environments, however, communication is inherently unreliable. Thus, message latency is highly unpredictable and can impact performance as much as disk latency. As another example, new applications such as reliable messaging rely chiefly on altering agents, whereas distributed TP relies chiefly on retrieving agents.

There is no question that new, application-specific commit protocols are needed. Until they emerge, my condition sets can help you determine when it's wise to recycle a TP commit protocol for your application, and when you might better get by with a simpler choice.

References

1. S. Gray and R. Lievano, *Microsoft Transaction Server 2.0,* SAMS Publishing, Indianapolis, Ind., 1997.
2. P.A. Bernstein and E. Newcomer, *Principles of Transaction Processing,* Morgan Kaufmann, San Mateo, Calif., 1997.
3. P.K. Chrysanthis, G. Samaras, and Y.J. Al-Houmaily, "Recovery and Performance of Atomic Commit Processing in Distributed Database Systems," V. Kumar, ed., *Recovery Mechanisms in Database Systems*, Prentice Hall, Upper Saddle River, N.J., 1998, pp. 370–416.
4. J. Gray, "A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem," *Fault Tolerant Distributed Computing*, Springer Verlag, Berlin, 1990, pp. 10–17.
5. S. Luan and V.D. Gligor, "A Fault-Tolerant Protocol for Atomic Broadcast," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 271–285.
6. R. Panadiwal and A.M. Goscinski, "A High Performance and Adaptive Commit Protocol for a Distributed Environment," *ACM Operating Systems Review*, Vol. 30, No. 3, July 1996, pp. 52–58.
7. J. Ouyang and G. Heiser, "Checkpointing and Recovery for Distributed Shared Memory Applications," *Proc. Fourth Int'l Workshop on Object-Orientation in Operating Systems (IWOOOS'95)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 191–199.
8. T. Kempster, C. Stirling, and P. Thanisch, "A Critical Analysis of the Transaction Internet Protocol," *Proc. Second Int'l Conf. Telecommunications and Electronic Commerce (ICTEC'99),* 1999; www.dcs.ed.ac.uk/home/tdk/projectpage.html (current Nov. 2000).
9. C.S. Li, C.J. Georgiou, and K.W. Lee, "A Hybrid Multilevel Control Scheme for Supporting Mixed Traffic in Broadband Networks," *IEEE J. Selected Areas in Communications*, Vol. 14, No. 2, Feb. 1996, pp. 306–316.
10. C. Mohan and D. Dievendorff, "Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queuing," *Data Engineering Bulletin*, Vol. 17, No. 1, Mar. 1994, pp. 22–28.
11. S. Mehrotra, H. Kexiang, and S. Kaplan, "Dealing with Partial Failures in Multiple Processor Primary-Backup Systems," *Proc. Sixth Int'l Conf. Information and Knowledge Management (CIKM'97)*, ACM Press, New York, 1997, pp. 371–378.
12. X. Liu, A. Helal, and W. Du, "Multiview Access Protocols for Large-Scale Replication," *ACM Trans. Database Systems*, Vol. 23, No. 2, June 1998, pp. 158–198.
13. I. Keidar and D. Dolev, "Increasing the Resilience of Distributed and Replicated Database Systems," *J. Computer and System Sciences*, Vol. 57, 1998, pp. 309–324.
14. P. Bizzarri, A. Bondavalli, and F. Di Giandomenico, "A Scheduling Algorithm for Aperiodic Tasks in Distributed Real-Time Systems and Its Holistic Analysis," *Proc. IEEE Future Trends in Distributed Computing Systems (FTDCS'97)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 296–301.

Peter Thanisch is a software designer at Accrue Software, Inc. His current interests include distributed protocols and online analytical processing. He received a PhD and MSc in computer science from the University of London. Readers can contact him at Accrue Software, Unit 10 Alpha Centre, Stirling University Innovation Park, Stirling FK9 4NF, Scotland; peter.thanisch@accrue.com.