Group Formation Mechanisms for Transactions in Isis

Neel K. Jain
Department of Computer Science
Cornell University
Ithaca, NY 14853
jain@cs.cornell.edu

Abstract

Distributed toolkits like Isis provide means of replicating data but not means for making it persistent. This makes the use of transactions desirable, even in non-database applications. Using Isis can alleviate the programming cost of distributed transaction processing and multi-phase commit protocols. Using the Isis transaction tool, however, imposes additional cost, and we examine the effect of group formation strategies on the overhead. The paper presents three different group formation mechanisms in Isis and compares the costs associated with them.

1 Introduction

A fundamental property of transactions is failure atomicity: transactions should leave a permanent effect if and only if they terminate normally (are committed) [2]. This property makes transactions appealing in both database and non-database applications, and many developers of general purpose operating systems advocate the use of transactional mechanisms in non-database contexts. However, multi-phase termination protocols used to implement atomicity can add significant programming and run-time overhead to distributed applications, requiring programming effort and computation and communication time. The two-phase commit protocol is often used to guarantee consistency in termination, but without a consistent timeout/failure-detection mechanism it can not guarantee progress if omission or crash failures occur. Moreover, in situations where data is replicated (notably in Isis, where data is replicated in process groups) serialization of transactions with respect to failure notification (as discussed in chapter 8 of [2]) is a potentially complex task.

The task of terminating a distributed transaction with a multi-phase commit protocol is made easier with the use of a distributed toolkit like Isis. The toolkit provides a collection of higher-level mechanisms for forming and managing process groups under an execution model called *virtual synchrony* [4]. Virtually synchronous execution can be used to guarantee serializability, even in the presence of failures [7].

Isis also provides a transaction tool for facilitating transactions. The performance of transaction processing in Isis, however, is limited by the suitability of Isis for implementing transactional commit, the key issue being the run-time cost. The current implementation of the Isis transaction tool uses a powerful, general purpose group formation mechanism to form a group of processes involved in the termination protocol, and this mechanism is one reason for the high cost of using the transaction tool. A group created solely to help terminate a transaction need not be created through this mechanism, and we examine two alternate strategies below, both of which show significant performance improvement over the original without sacrificing any consistency or progress guarantees.

1.1 Transactions and two-phase commit

A two-phase commit protocol can be used to terminate transactions atomically. The commit phase of such a transaction is divided into two phases (hence the name). In the first phase, the coordinator asks all participants to prepare to commit. All participants respond by returning a vote on whether the transaction should be committed. The coordinator decides that the transaction is to be committed if and only if it receives "yes" votes from all participants. In the second phase, the coordinator sends out its decision to all participants, which commit or abort their local transaction.

The commit protocol guarantees that a transaction will be committed only if all participants vote to commit, and that no participant will abort if another has committed (and vice-versa). Undetected failures may, however, inhibit the progress of one or more participants.

The undetected failure of the coordinator, for instance, may mean that a vote is never called, and leaves the participants waiting. The failure of a participant, if undetected, may leave the coordinator waiting for its vote. Communication failure can lead to the same situations. Below, we see how Isis, which provides reliable communication and failure detection, can be used to terminate a transaction in a consistent manner and prevent these situations.

1.2 Isis

Isis is a toolkit for building distributed fault-tolerant applications. It provides reliable communication, process monitoring and atomic multicast. Processes can be organized into *process groups*. The membership of an Isis group is available to any member, and all changes to the membership are ordered with respect to message delivery. Thus a

CIKM '94- 11/94 Gaitherburg MD USA

message is received in the same *group view* by all recipients.

Isis groups are normally formed by processes joining them by calling *pg_join*. When a process requests to join a group in this manner, the oldest member calls a *flush*, ensuring that all messages outstanding in the group are received by all (existing) members. Then an updated group view, with the new member, is sent out to the enlarged group. The flush algorithm and other Isis protocols are discussed in [3].

A group may also be formed from a list of process addresses by calling *pg_create*. The call creates the entire group and notifies the members. This mechanism is cheaper than repeated calls to pg_join, but requires that the membership be known in advance.

Isis provides several atomic multicast primitives. An Isis *cbcast* is a causally ordered multicast. *Abcasts* are totally ordered with respect to each other, and *gbcasts* are totally ordered with respect to all messages (and so, can be used to flush messages). Isis also provides a *reply* mechanism for each of these multicasts. The Isis manual [6] provides detailed information on Isis.

1.3 Transactions in Isis

Isis applications often use process groups to replicate data. Isis multicasts may be used to implement atomicity of updates, but not persistence of data, a need transactions can be used to fulfill. Implementing transactions in Isis is straightforward, since virtual synchrony may be used to provide serializability, and a multi-phase commit, atomicity.

The set of participants in a transaction is a natural candidate for an Isis group. The two-phase commit protocol can be implemented as two cbcasts in Isis. The first cbcast is a notice to participants to prepare to commit, and participant votes are sent as replies to this. The second cbcast is the decision to commit or abort.

Locking for transactions may be implemented using Isis abcasts, or with the use of the Isis token tool.

There are other benefits to using Isis groups for transactions as well: the failure of a participant can be detected using monitors, the list of participants is readily available (if needed), the coordinator may be replicated, and outcomes of transactions logged by Isis.

Isis also provides a tool for marking the start of a transaction: the activity ID. Such IDs are unique and last until a new one is generated. A message sent under an ID colors the receiver so that the receiving thread takes on the ID as well, that is, sends all future messages under this ID (figure 1). This frees the user from the requirement of explicitly marking every event in a transaction; any event that causally "happens after" the ID generation can be part of the transaction, and every process that has taken on the ID is, potentially, a participant. In fact, all potential participants take on the ID because every event causally after the ID generation is performed under that ID. However, the application is free to decide which of these processes should participate in the termination protocol. An interesting and useful feature of this mechanism is that the application need not keep track of which processes are involved in the transaction, or even who is coordinating the transaction. It need only decide when a transaction starts and when it ends, and whether a potential participant should participate in the termination protocol. Isis can take on the responsibility of forming the "transaction group", the group of participants in the transaction that will take part in the termination process. When the user-application decides that the transaction can be terminated it can issue a cbcast to this transaction group to prepare to commit, receive votes (as replies) and then, cbcast the decision to commit or abort.

2 The current transaction tool: using pg_join

It is not clear, however, how the transaction groups should be formed. Isis provides a transaction tool, and it uses activity IDs to mark transactions and to generate the name of the transaction group. On the receipt of a transaction-related message, the recipient of a message can use the activity ID to find the name of the transaction group, and join it, if it wishes, by calling pg_join. When the transaction is terminated, the coordinator broadcasts to this group, asking it to prepare to commit.

This approach is the most flexible and powerful of those discussed here. The list of participants in the transaction is available to all members and any changes in the groups can be observed by all participants at any point in the transaction.

This is an outline of the transaction's progress using this approach, and is shown in figure 2:

1. Transaction starts with a new identifier.
2. Participant notices that the transaction has started and joins the transaction group identified by the activity ID. The group can be monitored for the failure of the coordinator.
3. Coordinator decides that the transaction has completed. It calls for votes and checks replies to make certain that all participants have replied and that no one has voted to abort.
4. Coordinator sends out the outcome of the transaction. All monitors are cancelled and the group deleted.

This approach is the costliest of the three approaches studied. The high cost comes from the individual calls to pg_join made by each participant. The join requests are made almost concurrently, but have to be performed sequentially, and each involves a flush. This is expensive, and the performance of the transaction tool suffers.

Two variations have been suggested in an attempt to reduce this cost, and are described below.

3 The lightweight group mechanism

Patterns of group membership often recur in an Isis environment, and are very likely to recur in transactions. The lightweight group facility [5] takes advantage of this recurrence by side-stepping the expensive joins and leaves of heavyweight groups. A lightweight group, in Isis, is a veneer over a superset heavyweight group. Messages to the lightweight group are delivered to this heavyweight group and discarded at processes that do not belong to the lightweight group. Changes to the membership of a lightweight group may require that the underlying heavyweight group be replaced (if the joining member does not belong to the heavyweight group or the lightweight group becomes too small to use the heavyweight group efficiently), or even that a new one be created (if no suitable replacement can be found). If group membership patterns recur with regularity, a stable core of heavyweight groups will be created and members can be added to or removed from lightweight groups cheaply. After the system has stabilized, changes to the underlying set of heavyweight groups will be infrequent, even if the membership of the lightweight group changes frequently.

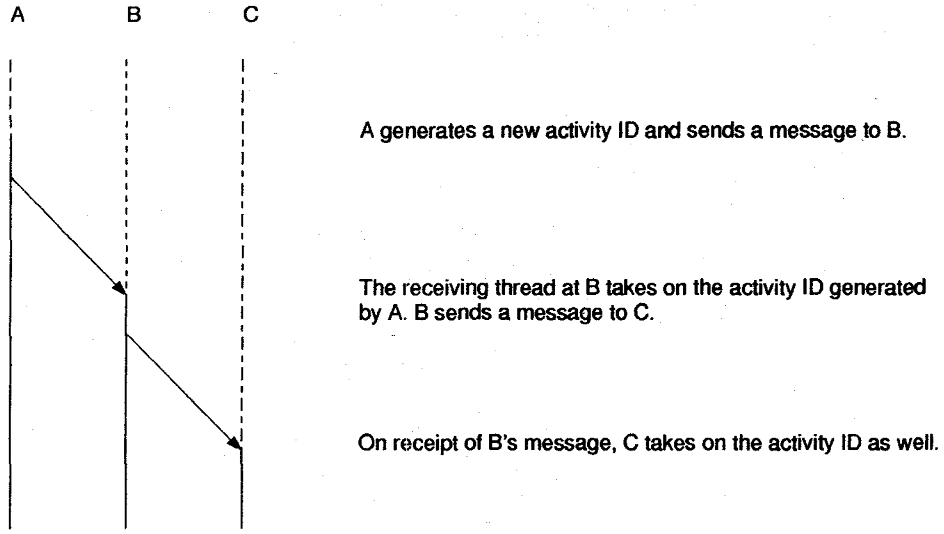

A
B
C
A generates a new activity ID and sends a message to B.
The receiving thread at B takes on the activity ID generated by A. B sends a message to C.
On receipt of B's message, C takes on the activity ID as well.

Figure 1: The transmission of Activity IDs in Isis. The solid line marks a thread colored with the new activity ID generated by A.

A lightweight group has all the functionality of a heavyweight group and a transaction tool based on lightweights groups will look similar to the Isis transaction tool. The membership is still available for any process to inspect and changes to it are ordered. One disadvantage to this approach, however, is that the cost of using lightweight groups depends heavily on the regularity of the recurrence of group membership patterns. If group membership patterns vary greatly, performance can drop to the level of that of regular groups, or even lower.

The outline of a transaction is exactly the same as in the previous case.

4 The lazy group formation mechanism

Both options considered so far maintain and can provide information about the membership of the transaction group at all times. Not all applications, however, need this information. Applications in which a participant does not need to know the list of co-participants, or does not need the list until voting has started, can use a third option: lazy group creation.

This variation is useful when a group is not needed until the coordinator enters the commit phase. Instead of doing a pg_join themselves, participants can send their address to the coordinator, which can form a list of addresses and create a group out of them at commit time, using the *pg_create* primitive.

The participants can monitor the coordinator for failure and abort the transaction if its failure is observed before the commit phase ends, that is, before the decision to commit or abort is sent out by the coordinator.

This is an outline of the transaction's progress using this approach (figure 3:

1. Transaction starts. Coordinator uses the transaction ID to generate a group name and joins this group. This group serves to identify the coordinator.
2. Participant notices that the transaction has started and sends address to the coordinator (via the group the coordinator has just created). Starts monitoring coordinator.
3. Coordinator collects all addresses and forms a group from them at commit time. Compares group size to list to check if any participants have failed (in which case the transaction is aborted).
4. Coordinator calls for votes and counts replies to make certain that all participants have replied and that no one has voted to abort.
5. Coordinator sends out the outcome of the transaction. Monitors are cancelled and the two groups created for the transaction are deleted.

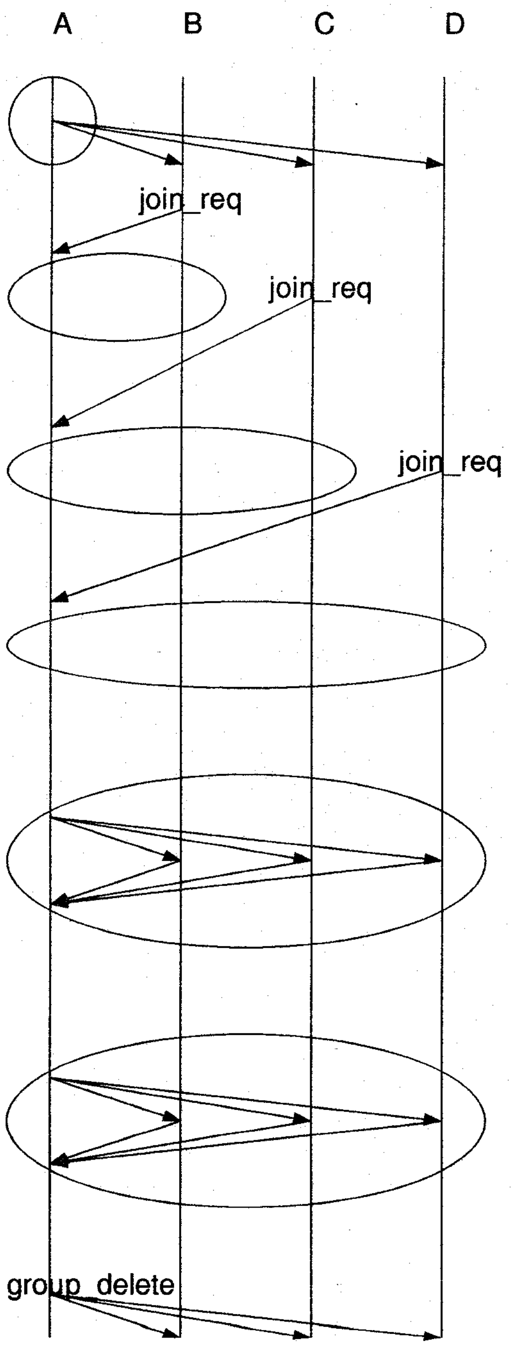
The approach uses two groups for each transaction. The first group is used for replying to the coordinator and monitoring it for failure. Although this adds to the cost of a transaction, it allows more than one transaction to proceed in the system, each transaction with its own, uniquely named, coordinator.

A disadvantage to the lazy group creation approach is that the failure of a participant is not detected until commit time. Thus, this approach may take longer than the original if a failure occurs.

5 Dealing with Failures

One of the attractions of Isis is its suite of failure-detection and monitoring mechanisms [6]. Handling failures during transactions is relatively straightforward, given the guarantees that Isis mechanisms provide and the "virtual synchrony" model of execution.

The easiest failure case to handle is the one in which only participants fail during a transaction. Handling the failure of the coordinator is slightly more difficult. The hardest case to handle is when participants fail with the coordinator, after votes have been cast but before the result has been announced to all participants. We discuss the various failure cases below. The actions suggested are not offered as the



Coordinator (A) creates the transaction group. Messages are sent out in the context of the transaction.

B, C and D call pg_join (or lwg_join). Isis sends a join request to the oldest member, A. The caller is then added to the group. Details of the join protocol are not shown.

The group is now complete.

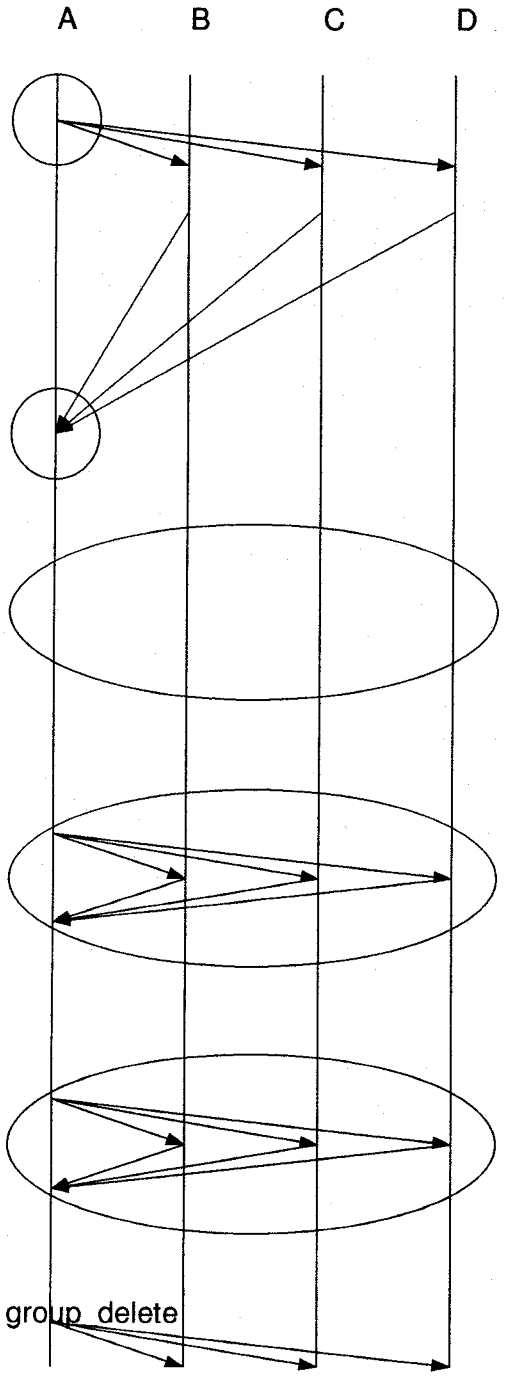
A starts the termination protocol by asking for votes (by a cbcast to the group). The votes are sent as replies.

A can now cbcast the result (and ask for acknowledgements if it so wishes).

A deletes the group. Isis sends a notification to the members of the group.

pg_join and lightweight group mechanisms

Figure 2: Transactions using the pg_join or lightweight group mechanism. The ovals represent Isis groups.



Coordinator creates an identifying group. Messages are sent out in the context of the transaction.

B, C and D send their addresses to the coordinator, A.

A creates the transaction group from the addresses received.

A starts the termination protocol by asking for votes (by a cbcast to the group). The votes are sent as replies.

A can now cbcast the result (and ask for acknowledgements if it so wishes).

A deletes the group. Isis sends a notification to the members of the group.

lazy group creation

Figure 3: Transactions using the lazy group creation mechanism. The ovals represent Isis groups.

only, nor as the most efficient, course that can be taken on noticing a failure; they merely illustrate one of the several consistent ways to handle failures.

5.1 Before voting starts

As long as the coordinator does not fail, it can provide a canonical view of the transaction, no matter which approach we take to group formation. For this reason, participants need only monitor the coordinator, and abort the transaction if the coordinator fails before all participants have voted. Participants in the pg_join and LWG approaches monitor the transaction group as soon as they join it. If the coordinator fails, another process finds itself the oldest member of the group (this can also happen as soon as the process joins the group). This process can then abort the transaction and delete the group. Any process joining the group after this will find itself the oldest member and do the same.

Participants in the lazy group creation approach monitor the coordinator explicitly. This monitor, however, does not provide strong ordering guarantees, not having been set up in the context of a group, so if participants are informed of the coordinator's failure, they need to check if the transaction group has been formed, and flush it if it does (to check if a call for votes had been issued, in which case they can try to salvage the transaction as discussed below).

Once a call for votes has been issued in any of the three approaches, a transaction group exists, and failure detection and handling is identical in all three cases.

5.2 Once voting starts

Before the coordinator issues a cbcast for votes, it knows how many votes it expects to get, and can send this number with its request for votes. If it receives fewer than this number of votes as a result of the cbcast, it can abort the transaction. If all votes are received, the coordinator can examine them, send a message with the result, and ask for replies. If it receives fewer replies than wanted, it can log the outcome of the transaction (using a presumed-commit/abort strategy if it likes).

If the coordinator fails after it has asked for votes, the oldest surviving participant can take over and try to recollect votes (this assumes that the coordinator was in favor of committing if it asked for votes). If the new coordinator can collect the required number of votes, it can ask the group to commit or abort. Of course, any member that had not returned a vote in response to the original call can now send a vote to abort, allowing the new coordinator to abort the transaction. In case the new coordinator cannot muster the required number of votes, and no participant votes to abort, the group must block.

This is the case where the coordinator and at least one more participant have crashed and the remaining members are unable to tell if (and how) this participant had voted. Worse, this participant could have received the outcome from the coordinator and acted on it. In this case, blocking is the only choice available to the survivors.

5.3 Non-blocking strategies

Two-phase commit protocols cannot guarantee that they will not block [2]. However, there is a (hidden but usable) third phase to our termination protocol: the coordinator deletes the group at the end of the transaction by calling pg_delete. The deletion notification is ordered with respect to all messages. If the coordinator waits for the completion of the cbcast that sends the outcome to the group (by any of several Isis mechanisms, including asking for replies) before it calls pg_delete, a non-blocking protocol can be fashioned by modifying the protocol above.

The protocol is identical to the one already discussed up to the point that the outcome is received. The participants receive the outcome but do not act on it until they are informed of either the deletion of the group, or the failure of the coordinator. If they are notified of the group deletion first, they are free to act on the outcome and cancel outstanding watches and monitors. If they notice the coordinator crash before observing the delete, they cbcast the outcome to the group, asking for acknowledgments, then act on the outcome. This ensures that every surviving member of the group has seen the outcome before any of them acts on it.

6 The implementation

The implementation studied was a no-frills implementation, with no logging or recovery mechanisms. There was no transaction to speak of; once the group was formed, the coordinator started the voting process. The goal of the implementation was to highlight the differences between the group formation mechanisms, so no attempt was made to include logging or recovery mechanisms.

All processes started by joining a large, main group. The coordinator started a transaction by a cbcast to the group and waited for replies. In the case of the regular group mechanism the receivers joined the transaction group before replying to the coordinator. In the lazy group creation case, the receiver replied to the cbcast with its address. Monitors were set up before replies were sent.

Next, the coordinator started the voting process by a cbcast to the transaction group (creating the group first, in the lazy case). The recipients replied with their vote (the ones in the lazy case first changing the monitor from a process monitor to a group monitor with stronger guarantees).

The coordinator then tallied the votes and cbcasted the results. Monitors were cancelled on the receipt of the tally. The group was deleted by the coordinator on the completion of the cbcast, and data structures relating to the last transaction removed.

The coordinator was run on a site by itself. For the lightweight group case, we also ran the gvmgr process (which provides group view services for the lightweight group implementation used) on the same site. We then ran one participant on each of two remote sites, one participant on each of four remote sites, two participants on each of three remote sites and two participants on each of four remote sites. Each of these sites had Isis running locally.

6.1 The results

The lazy group creation approach proved to be the cheapest of all three approaches for group sizes of six or more. The lightweight group approach was slightly more expensive for large groups, but cheaper for smaller ones. The regular, pg_join approach was far more expensive.

For group sizes of three (the coordinator and two other participants), and five (coordinator and four other participants) the lightweight group mechanism was cheaper than the create mechanism. This is likely a result of the small number of processes involved (less book-keeping overhead), and the advantage disappears as the group size increases.

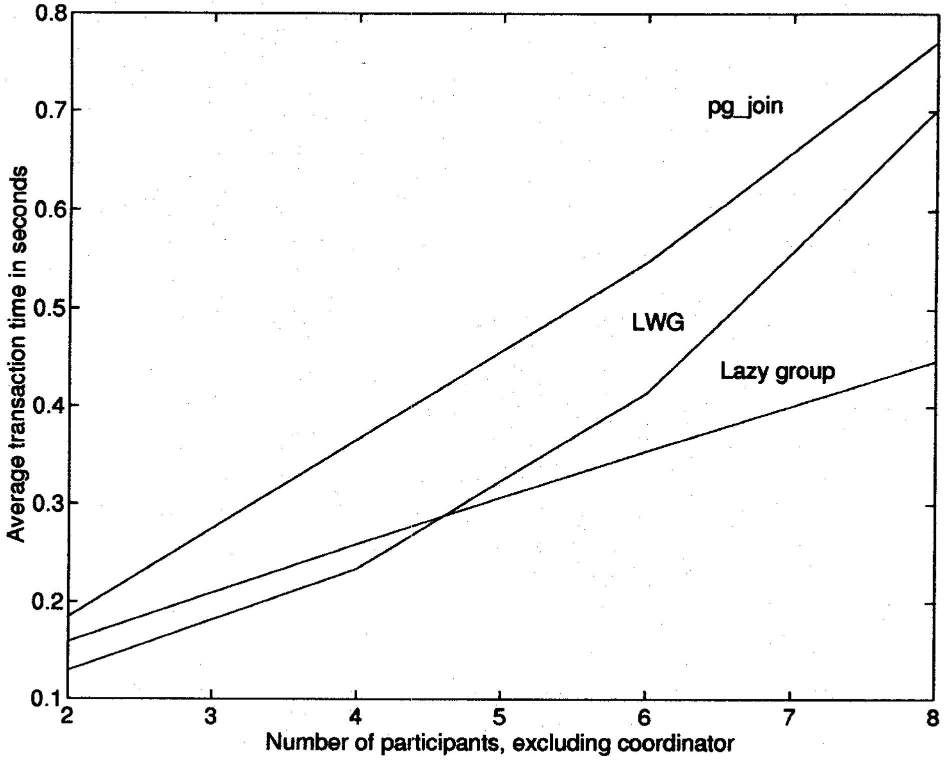


Figure 4: A comparison of average time for transactions with 2, 4, 6 and 8 participants, in addition to the coordinator. Times are averaged over 1000 transactions.

The lazy group creation approach is hindered by having to create two groups for each transaction, but the significance of this extra cost decreases with the size of the group. As the group size increases to seven and nine, we see that the lazy creation approach gains an advantage over the lightweight group mechanism. Average times for the three approaches are compared in figure 4.

7 Related Work

The paper's discussion and results have, so far, centered on Isis. Other distributed computation toolkits, notably Transis [1], also provide group mechanisms and reliable communication. The Transis approach differs from that of Isis in some respects, but it provides all the functionality required for distributed transaction processing. Transis also provides virtual synchrony, assuring that the execution model is powerful and sound.

A feature of Transis not available in Isis is *safe* messages. Such messages are delivered to a recipient only when the message has been received (but not necessarily delivered) at all other recipients. A non-blocking protocol can be designed using safe messages to announce the coordinator's decision to commit or abort.

8 Conclusion

Transactional mechanisms guarantee serializability and failure atomicity, and so, address the needs of even non-database applications that require persistent data. Applications using distributed environments like Isis, where data may be replicated, but only in a volatile manner, find it useful to implement transactions.

When data is replicated, or the transaction is distributed across several processes, serializability and atomicity are difficult to implement. Isis provides a framework for consistent handling of both issues. Implementations for distributed transaction processing in Isis can use a transaction group to keep track of processes involved in a transaction. The task of forming the transaction group and terminating the transaction is simplified by the mechanisms provided in Isis, but the cost of using the Isis transaction tool may be prohibitively high. One of the reasons behind this high cost seems to be the group formation mechanism used by the tool, and we have studied two alternatives to this mechanism, each with its own strengths and weaknesses.

The current pg_join approach is the most flexible and powerful, but also the slowest. The lightweight group approach is also flexible and powerful, and significantly faster than the pg_join approach for recurring group patterns. The third approach, lazy group creation, is the fastest for large group sizes, but sacrifices some flexibility. We have seen, however, that any of the three approaches is sufficient to handle transactions with little effort. An application should consider its transaction patterns and requirements to choose between these mechanisms.

9 Acknowledgments

Ken Birman and Brad Glade provided invaluable comments on drafts of this paper. Brad Glade also assisted with Isis. Robert Cooper provided the implementation of, and help with, lightweight groups. Thomas Yan provided comments on an early draft.

Research on Isis was supported by the Department of Defense under ARPA/ONR grant N00014-92-J-1866. The work reported in this paper was motivated during a summer internship with the Open OODB Project at Texas Instruments' Computer Sciences Laboratory, Dallas, Texas.

References

[1] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, Massachusetts, July 1992. IEEE.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[3] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *Transactions on Computer Systems*, pages 272–314, August 1991.

[4] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, pages 37–53, December 1993.

[5] Bradford B. Glade, Kenneth P. Birman, Robert C. B. Cooper, and Robbert van Renesse. Light-weight process groups in the Isis system. *Distributed Systems Engineering*, pages 29–36, July 1993.

[6] The Isis Group. *The Isis Distributed Toolkit Version 3.0 User Reference Manual.* Department of Computer Science, Cornell University, 1991.

[7] T. Joseph and K. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *Transactions on Computer Systems*, 4(1):54–70, February 1986.