DYNAMIC COMMIT TREE MANAGEMENT FOR SERVICE ORIENTED ARCHITECTURES

Stefan Böttcher and Sebastian Obermeier
University of Paderborn, Computer Science, Fürstenallee 11, 33102 Paderborn, Germany

Keywords: Service oriented architectures, SOA, web services, web service transactions, atomicity, commit tree.

Abstract: Whenever Service Oriented Architectures make use of Web service transactions and an atomic processing of these transactions is required, atomic commit protocols are used for this purpose. Compared to traditional client server architectures, atomicity for Web services and Web service composition is much more challenging since in many cases sub-transactions belonging to a global transaction are not known in advance.
In this contribution, we present a dynamic commit tree that guarantees atomicity for transactions that invoke sub-transactions dynamically during the commit protocol's execution. Furthermore, our commit tree allows the identification of obsolete sub-transactions that occur if sub-transactions are aborted and restart.

1 INTRODUCTION

An atomic transaction execution is essential for concurrent transactions in complex systems like distributed databases, peer-to-peer systems, and Service Oriented Architectures that make use of a composition of cascading Web service calls.

Service Oriented Architecture that use Web service transactions demand a more dynamic transactional model than the classical distributed transactional model for which 2PC is designed. More precisely, the model must support a dynamic invocation of sub-transactions even during commit time. This means, that the use of classical atomic commit protocols like 2PC (Gray, 1978) or 3PC (Skeen, 1981) involves a lot of problems since a previously unknown set of sub-transactions must be guided to an atomic commit decision.

Solutions based on timeouts (Kumar et al., 2002) demand committed transactions to be undone by applying compensation transactions. However, committed transactions can trigger other operations, and we cannot assume that compensation is always possible for the following reason. In networks where network partitioning can occur, nodes may be unreachable but still operational. Therefore, compensation transactions may not reach a separated, fully operational node. Meanwhile, this node may have executed other transactions based on the transaction that should have been compensated. To avoid these kinds of problems, we focus on a transaction model, within which atomicity is guaranteed for distributed, non-compensatable transactions.

In this paper, we propose a data structure used by a commit coordinator to deal with dynamically invoked sub-transactions. The data structure re-organizes itself and allows the coordinator to react to various network situations.

2 TRANSACTION MODEL

Our transaction model has the goal to support atomic execution of Web services. Our transaction model is based on the concepts "application", "transaction procedure", "Web service", and "sub-transaction", as well as their relationship to each other.

An *application AP* may consist of one or more *transaction procedures*. A transaction procedure is a Web service that must be executed in an atomic fashion. Transaction procedures and Web services are implemented using local code, database instructions, and (zero or more) calls to other remote Web services. Since the invocation of a Web service depends on conditions and parameters, different executions of the same Web service may call different Web services and execute different local code.

We call the execution of a transaction procedure a global transaction T. The application *AP* is only interested in the result of T, i.e. whether the execution of a global transaction T has been committed or aborted. In case of commit, *AP* is also interested in the return values of the parameters of T.

The relationship between transactions, Web services, and sub-transactions is recursively defined as follows: We allow each transaction or sub-transaction T to dynamically invoke additional Web services offered by physically different nodes. We call the execution of such Web services invoked by the transaction T or by a sub-transaction T_i the sub-transactions $T_{si} \ldots T_{sj}$ of T or of T_i, respectively. This invocation hierarchy can be arbitrarily deep.

Whenever $T_1, \ldots, T_n$ denote all the sub-transactions called by either T or by any child or descendant sub-transaction T_s of T during the execution of the global transaction T, atomicity of T requires that either all transactions of the set $\{T, T_1, \ldots, T_n\}$ commit or all of these transactions abort.

We assume that each Web service only knows the Web services that it calls directly, but not whether or not the called Web services call other Web services. Therefore, at the end of its execution, each transaction T_i knows which sub-transactions $T_{is_1} \ldots T_{is_j}$ it has called, but T_i, in general, will not know which sub-transactions have been called by $T_{is_1} \ldots T_{is_j}$. Furthermore, we assume that usually a transaction T_i does not know how long its sub-transactions $T_{is_1} \ldots T_{is_j}$ are going to run.

We assume that each sub-transaction consists of the following phases: a read-phase, a coordinated commit decision phase, and, in case of successful commit, a write-phase. During the read-phase, each sub-transaction performs write operations on its private storage only. After commit, during the write phase, write operations on the private storage are transferred to the database, such that the changes done throughout the read-phase become visible to other transactions after completion of the write-phase.

In the architecture for which our protocol is designed, Web services are invoked by messages instead of invoking them by a synchronous call to a Web service for the following reason. We want to avoid that a Web service T_i that synchronously calls a sub-transaction T_j cannot complete its read phase and cannot vote for commit before T_j sends its return value to T_i. Therefore, we allow sub-transactions only to return values indirectly by asynchronously invoking corresponding receiving Web services, and not synchronously by return statements[1]. Since (sub-) transactions describe general services, the nodes that execute these (sub-) transactions may be arbitrary nodes and are not necessarily databases. We therefore call these nodes *resource managers (RM)*.

One characteristic of our Web service transactional model is that the initiator and the Web services do not know every sub-transaction that is generated during transaction processing. Our model differs from other models that use nested transactions (e.g. (Dunham et al., 1997), (OMG, 2003), (Cabrera et al., 2005)) in some aspects including but not limited to the following:

- Since network partitioning makes it difficult or even impossible to compensate all sub-transactions, we consider each sub-transaction running on an individual resource manager to be non-compensatable. Therefore, no sub-transaction is allowed to commit independently of the others or before the commit coordinator guarantees that all sub-transactions can be committed.
- Different from CORBA OTS ((OMG, 2003), (Liebig and Kühne, 2005)), we assume that we cannot identify a hierarchy of commit decisions, where aborted sub-transactions can be compensated by executing other sub-transactions.
- Different from the Web service transaction model described in (Cabrera et al., 2005), the Initiator of a transaction in our model does not need to know all the transaction's sub-transactions. We assume that the Initiator is only interested in the commit status and the result of the transaction, but not in knowing all the sub-transactions that have contributed to the result.
- A Web service may consist of control structures, e.g. `if <Condition> then <T1> else <T2>.`This means that a sub-transaction executing this Web service may create other sub-transactions dynamically. These dynamically created sub-transactions also belong to the global transaction and must be executed in an atomic fashion.
- Communication is message-oriented, i.e., a Web service does not explicitly return a result, but may

[1]However, if the application needs synchronous calls, e.g. because of dependencies between sub-transactions, the intended behavior can be implemented by splitting T_i into T_{i_1} and T_{i_2} as follows. T_{i_1} includes T_i's code up to and including an asynchronous invocation of its sub-transaction T_j; and T_{i_2} contains the remaining code of T_i. T_j performs an asynchronous call to T_{i_2} which may contain return values computed by T_j that shall be further processed by T_{i_2}

invoke a receiving Web service that performs further operations based on the result.

3 SOLUTION

This section describes a data structure called “commit tree”, which represents the commit status of all sub-transactions involved in a global transaction.

3.1 Atomicity of Web Service Transactions

The main problem of ensuring atomicity for Web service transactions is that the coordinator does not know all sub-transactions. In order to inform the coordinator about all invoked sub-transactions, we outline the sub-transaction’s ID management in Section 3.2 and propose a data structure called *commit tree* to dynamically store the completion status of each sub-transaction involved in a transaction in Section 3.3.

3.2 The Sub-Transactions ID Management

Since a global transaction T may consist of many invoked sub-transactions, we propose the use of sub-transaction IDs to distinguish all sub-transactions of T. To ensure that the coordinator gets knowledge of all invoked sub-transactions belonging to T, we require that each vote message sent by a sub-transaction T_i to the coordinator informs the coordinator about all sub-transactions $T_{s_1},\dots,T_{s_k}$ that are called by T_i. This means that each sub-transaction T_i includes a list of IDs of all its invoked sub-transactions $T_{i_1},\dots,T_{i_k}$ in its vote message. For this purpose, we have included a parameter `ListOfInvokedSubTransactions` in the vote message which has the following format:

```
VoteMessage V(bool commit,
  ID subtransactionID,
  ID callerID, ID globalTID,
  ListOf(ID) ListOfInvokedSubTransactions,
  int sequenceNr)
```

Since the atomic decision of the global transaction T must include these invoked sub-transactions, the coordinator must also wait for the votes of those newly added resource managers. This behavior is supported by the commit tree data structure, for which an example is given in the next section and which is generally defined in Section 3.4.

The other parameters have the following meaning: `commit` tells the coordinator whether or not the sub-transaction execution was successful and contains the value of either “abort” or “commit”. `subtransactionID` is the sub-transaction’s own ID whereas `callerID` is the ID of the parent transaction and `globalTID` is the ID of the global transaction T to which the sub-transaction belongs. Finally, `sequenceNr` is needed to identify the latest version of the vote message in order to handle message delays if the vote message is sent more than once.

Furthermore, each sub-transaction T_{s_i} belongs to exactly one global transaction T, and it is called by exactly one caller T_s, i.e., the application program or another sub-transaction. Since each sub-transaction T_s must inform the coordinator about the sub-transaction IDs of all its sub-transactions $T_{s_1},\dots,T_{s_k}$, we have decided to provide a sub-transaction T_{s_i} with all the information when it is called. Therefore, the sub-transaction ID of T_{s_i} is generated by the resource manager running the calling parent transaction T_s and is passed in a parameter `subtransactionID` to T_{s_i} when T_{s_i} is invoked. The following example shows the use of IDs:

```
invokeSubTransaction( subtransactionID,
  callerID, globalTID,
     <WebService name and parameters>)
```

The parameter `callerID` contains the ID of T_{s_i} and the parameter `globalTID` contains the ID of the global transaction T to which T_{s_i} and T_s belong. The parameter list `<WebService name and parameters>` contains the name of the called Web service and the parameters used for the Web service call.

3.3 An Example of the Coordinators Commit Tree

To ensure that all sub-transactions $T_{s_i},\dots,T_{s_j}$ invoked by a sub-transaction T_i are known to the coordinator, the coordinator must process the parameter `ListOf-InvokedSubTransactions`, passed in the vote message of T_i and update the set of required votes for the global transaction T before processing T’s commit decision. Before we describe the general use of the commit tree data structure, we give an example (c.f. Figure 1) that shows how we ensure that the coordinator can only come to a commit decision after it has knowledge of all necessary votes.

Figure 1 shows part of a sequence diagram of an example execution: During the read-phase, a sub-transaction T2 needs the service doT(...) and generates the ID “T4” to invoke the Web service with the required parameters `subtransactionID` (T4), `CallerID` (T2) and `globalTID`. In our example, sub-transaction T4 has finished earlier than T2 and sends

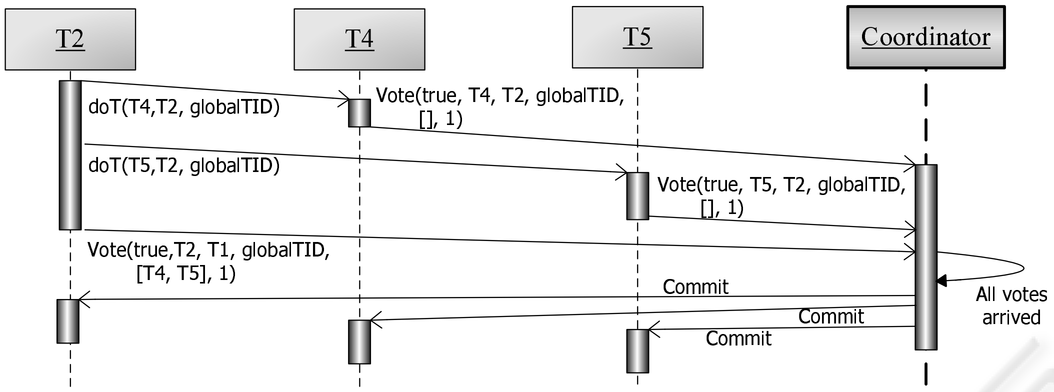
T2
T4
T5
Coordinator
doT(T4,T2, globalTID)
Vote(true, T4, T2, globalTID, [], 1)
doT(T5,T2, globalTID)
Vote(true, T5, T2, globalTID, [], 1)
Vote(true,T2, T1, globalTID, [T4, T5], 1)
Commit
Commit
Commit
All votes arrived

Figure 1: Sequence diagram of example commit tree construction.

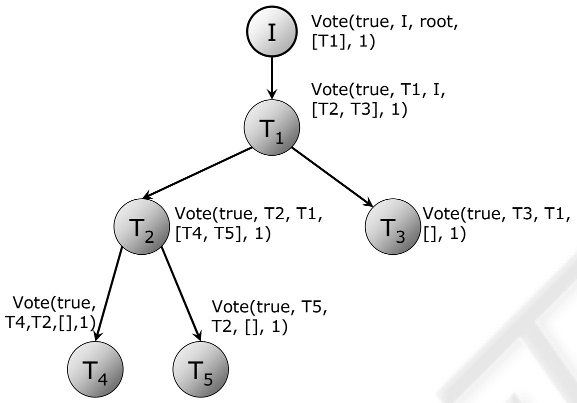
I
Vote(true, I, root, [T1], 1)
Vote(true, T1, I, [T2, T3], 1)
T_1
T_2
Vote(true, T2, T1, [T4, T5], 1)
T_3
Vote(true, T3, T1, [], 1)
Vote(true, T4,T2,[],1)
Vote(true, T5, T2, [], 1)
T_4
T_5

Figure 2: An example commit tree.

the vote message to the coordinator. Since T4 has not invoked any sub-transactions, the fifth parameter of the vote message denoting the set of the called sub-transactions is empty. T2, however, includes the IDs of its invoked sub-transactions, T4 and T5, in its set of the called sub-transactions. This makes the coordinator to require the votes of T4 and T5, which, in this example, have arrived earlier. When the coordinator has received all these votes, it decides on the global commit decision.

In contrast to 2PC, which handles flat transactions instead of nested transactions, the dynamic call of Web services supported by our protocol requires that the votes, which the coordinator needs for a global commit decision, can be determined only during the protocol's execution. In order to store the commit status and vote of each sub-transaction, we propose a dynamic data structure, called *commit tree*, and we define the initiator of a transaction to be the root of this commit tree.

Figure 2 shows an example commit tree (we have omitted the globalTIDs since all sub-transactions belong to the same global transaction). When the coordinator has received the initiator *I*'s vote which includes the list [T1] for the parameter of the invoked sub-transactions, the root node is generated to represent the commit status of *I*. When the coordinator has received the vote of T1, the node T1 is created. Since the sub-transaction T1 invoked the sub-transactions T2 and T3, the commit of the sub-transactions T2 and T3 is also required to commit the whole transaction. Therefore, this information is added to the commit tree. The coordinator builds this commit tree dynamically and determines whether all votes needed for continuing the protocol's execution have arrived. Since the information about invoked sub-transactions is sent along with a vote message of the parent transaction and the parent's vote can arrive later than a sub-transaction's vote, it may be the case that an arriving sub-transaction vote cannot be assigned to a parent transaction, like the vote messages of T4 and T5 in the example of Figure 1. In this case, the sub-transaction's vote is stored and it is assigned after the corresponding parent sub-transaction's vote has arrived.

3.4 The Coordinators Commit Tree

The commit tree for a transaction is an unordered tree with additional parameters to dynamically store votes. Each commit tree corresponds to exactly one global transaction and stores the following variables: the global transaction ID; a tree structure containing commit tree nodes; a list `unassignedN` of unassigned nodes corresponding to sub-transactions that voted for commit before their parents voted for commit; and a list openSubTransactions of transaction IDs, for which the vote was not received by the coordinator. Furthermore, each commit tree *node* stores: the sub-transaction ID; the ID of the resource manager running the sub-transaction; the parent transactionID; and $0-n$ IDs of invoked sub-transactions.

Depending on the status of the commit tree and based on a timer, the coordinator decides on the commit decision and sends one of the following messages to all participating resource managers:

doCommit, which requires all recipients to commit the sub-transaction. the message is sent, when all participants have voted for commit, i.e. the list openSubTransactions is empty.

doAbort, which requires the recipient to abort the sub-transaction. This message is sent, when at least one participant has voted for abort.

3.5 Modification of the Commit Tree by the Vote Operation

Algorithm 1 outlines the implementation of the coordinator's vote operation which is executed on the commit tree whenever a client's vote message arrives at the coordinator. First, the coordinator uses the sequence number to check that no newer message was processed earlier, e.g., due to message delay (line refalg1:check)[1]. Thereafter, a new node N is created and the reception of the vote is marked in the list of open sub-transactions (line 8). Then, the parent-child relationships between N and the nodes representing other sub-transactions are managed (lines 8-14). In addition, the implementation of the vote operation (line 15) updates the list `openSubTransactions` which stores sub-transactions of which the votes are necessary.

If all votes are present, the list `openSubTransactions` contains only marked entries and the global decision can be made. To ensure that in case of a resource manager's failure no infinite blocking occurs, the coordinator starts the timer. If the time is over and some votes are still missing, the coordinator may propose abort or it may inform the participants about the delay and give them the chance to file a petition to abort the transaction, e.g. if highly needed resources are blocked by the pending transaction. However, only the coordinator may abort the transaction after a resource manager voted for commit.

[1]This is necessary because otherwise an old message would overwrite a newer message, if the old message was delayed

4 RELATED WORK

We can distinguish contributions to the field of atomicity and distributed transactions according two main criteria: first, whether their transactional models use flat transactions or support nested transaction calls, second, whether transactions and sub-transactions are regarded as compensatable or non-compensatable. Our contribution is based on a transactional model that allows nested transaction calls and assumes sub-transactions to be non-compensatable.

The requirement to allow sub-transactions to invoke other sub-transactions originated from business applications. Within such a business application, the atomicity constraint is to complete all "sub-transactions" of a *workflow* (Workflow Management Coalition, 2000). Today, Web services and their description languages (e.g. BPEL4WS (Curbera et al., 2002) or BPML (Arkin et al., 2002)) are more and more used to implement nested Web service transactions, which are called *Web services orchestration*.

However, these languages do not provide a coordination framework to implement atomic commit protocols. For this purpose, our contribution can be combined with these description languages, like the "WS-Atomic-Transaction" proposal (Cabrera et al., 2005) does. Note that our contribution is different from (Cabrera et al., 2005) in several aspects. For example, (Cabrera et al., 2005) has a "completion protocol" for registering at the coordinator before the transaction coordination started. In comparison, our commit tree is invoked on-the-fly and can be dynamically extended, which is needed whenever used Web services are not known in advance.

Besides the Web service orchestration model, there are other contributions that set up transactional models to allow the invocation of sub-transactions, e.g. the Kangaraoo Model (Dunham et al., 1997). Common with these transaction models, we have a global transaction and sub-transactions that are created during transaction execution and cannot be foreseen. The main difference to this transactional model lies in the fact that we consider all sub-transactions to be non-compensatable. This means that in our model, an abort of one of these sub-transactions must – for the sake of the atomicity constraint – result in a global abort.

Other contributions like (Pitoura and Bhargava, 1995) or (Rakotonirainy, 1998) allow a transaction to define the level of consistency which the transactions leave behind. In contrast, our solution does not need to adjust the level of consistency. We can guarantee atomicity without leaving states of inconsistency, even within environments where nodes that have con-

Algorithm 1 Coordinator's implementation of the Vote procedure.

```
procedure VOTE(boolean commitStatus, ID subtrsID, ID callerID, ID globalTID, ListOf(ID) invokedSubT, int sequenceNr)
    if isVoteValid(sequenceNr) then                          ▷ no newer message was processed earlier?
        if commitStatus==false then
            abortTransaction();                              ▷ If one participants voted abort, abort the transaction
        end if
        N :=createNode(subtrsID, callerID, globalTID, invokedSubT);
        openSubTransactions.markAsVoted(subtrsID);
        if (ParentNode := getNodeByID(callerID)) == nil then
            unassignedN.add(N);                              ▷ Vote of the calling sub transaction has not yet arrived
        else
            ParentNode.addChild(N)
            assignNodes(invokedSubT, N)
        end if
        openSubTransactions.add(invokedSubT)
    end if
end procedure
```

sistent data may crash or permanently remain in a separated network partition.

Corba OTS ((OMG, 2003), (Liebig and Kühne, 2005)), uses a hierarchy of commit decisions, where an abort of a sub-transaction does not necessarily lead to an abort of the global transaction. Instead, the calling sub-transactions can react on this abort and use other sub-transactions, for example. Although we also assume that Web services invoke other Web services and the coordinator uses a tree structure to maintain information about commit votes, we do not have this hierarchical commit decisions, since in the presence of non-compensability, this implies waiting for the commit decision of all descendant nodes. In an environment where node failures are likely, we do neither propose to wait nor to block the participants until the commit decision has reached all participants. In contrast, our solution needs only one message round for distributing the commit decision to each participant.

When the coordinator's availability cannot be guaranteed throughout the whole transaction execution, for example in mobile environments where disconnections and network failures may occur, some contributions propose the use of protocols with more than one coordinator. Since our solution is an extension, it can be combined with these protocols. (Reddy and Kitsuregawa, 2003), for example, suggests the use of backup coordinators in 2PC; (Gray and Lamport, 2004) uses Paxos Consensus (Lamport, 1998) to get a consensus on the commit decision; and (Böse et al., 2005) allows "controlled failures" by proposing a combination of 2PC, 3PC (Skeen, 1981), and Paxos Consensus. Different contributions like (Nouali et al., 2005) employ *participant-agents* to shift the coordination workload to fixed, stable parts of the network like base stations. However, our extension is also suitable in such infrastructure scenario. Since all these referenced protocols require a resource manager to send a vote on the sub-transaction, the protocols can be extended to support the dynamic invocation of sub transactions within a service oriented architecture.

5 SUMMARY AND CONCLUSION

In this paper, we have presented the commit tree as a key idea for guaranteeing atomicity for Web service transactions. The commit tree, a specialized data structure, can be used to implement the coordinator's management of transaction atomicity for a dynamically changing set of sub-transactions. We have embedded our atomic commit protocol in a Web service transactional model, the characteristics of which is that sub-transactions must not be known in advance. Finally, our protocol extension merges nicely with a variety of concurrency control strategies including validation and locking.

REFERENCES

Arkin, A. et al. (2002). Business process modeling language, bpmi.org. Technical report.

Böse, J.-H., Böttcher, S., Gruenwald, L., Obermeier, S., Schweppe, H., and Steenweg, T. (2005). An integrated commit protocol for mobile network databases. In *9th International Database Engineering & Application Symposium IDEAS*, Montreal, Canada.

Cabrera, L. F., Copeland, G., Feingold, M., et al. (2005). Web Services Transactions specifications – Web Services Atomic Transaction. http://www-

128.ibm.com/developerworks/library/specification/ws-tx/.

Curbera, F., Goland, Y., Klein, J., Leymann, F., et al. (2002). Business Process Execution Language for Web Services, V1.0. Technical report, BEA, IBM, Microsoft.

Dunham, M. H., Helal, A., and Balakrishnan, S. (1997). A mobile transaction model that captures both the data and movement behavior. *Mobile Networks and Applications*, 2(2):149–162.

Gray, J. (1978). Notes on data base operating systems. In Flynn, M. J., Gray, J., Jones, A. K., et al., editors, *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer.

Gray, J. and Lamport, L. (2004). Consensus on transaction commit. *Microsoft Research – Technical Report 2003 (MSR-TR-2003-96)*, cs.DC/0408036.

Kumar, V., Prabhu, N., Dunham, M. H., et al. (2002). Tcot-a timeout-based mobile transaction commitment protocol. *IEEE Trans. Comput.*, 51(10):1212–1218.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Liebig, C. and Kühne, A. (2005). Open Source Implementation of the CORBA Object Transaction Service. http://xots.sourceforge.net/.

Nouali, N., Doucet, A., and Drias, H. (2005). A two-phase commit protocol for mobile wireless environment. In Williams, H. E. and Dobbie, G., editors, *Sixteenth Australasian Database Conference (ADC2005)*, volume 39 of *CRPIT*, pages 135–144, Newcastle, Australia. ACS.

OMG (2003). Transaction Service Specification 1.4. http://www.omg.org/.

Pitoura, E. and Bhargava, B. K. (1995). Maintaining consistency of data in mobile distributed environments. In *International Conference on Distributed Computing Systems*, pages 404–413.

Rakotonirainy, A. (1998). Adaptable transaction consistency for mobile environments. In *DEXA Workshop*, pages 440–445.

Reddy, P. K. and Kitsuregawa, M. (2003). Reducing the blocking in two-phase commit with backup sites. *Inf. Process. Lett.*, 86(1):39–47.

Skeen, D. (1981). Nonblocking commit protocols. In Lien, Y. E., editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan*, pages 133–142. ACM Press.

Workflow Management Coalition (2000). http://www.wfmc.org/.