

# A Dynamic Two-Phase Commit Protocol for Self-Adapting Services

Weihai Yu      Yan Wang

*University of Tromsø*  
*Department of Computer Science*  
*N-9037 Tromsø, Norway*  
*{weihai,wang}@cs.uit.no*

Calton Pu

*Georgia Institute of Technology*  
*College of Computing*  
*Atlanta, GA 30332-0280, USA*  
*calton@cc.gatech.edu*

## Abstract:

*Next-generation applications based on Web services impose additional requirements on the use of coordination protocols with various optimizations, such as the two-phase commit protocol (2PC). This paper analyses the well-known 2PC optimizations presumed commit and presumed abort, and presents an improved 2PC that is suitable for Web services based applications. More specifically, the protocol allows every individual service provider to choose dynamically the most appropriate presumption for any distributed transaction. The protocol does not introduce extra overhead to the previous 2PC variants in terms of number of messages and log records, and it is easy to understand and realize.*

## 1. Introduction

One of the goals of Web services is to enable next-generation applications with dynamic interactions among data services. Consider for instance a virtual manufacturer consisting of a chain of distributed outsourced services such as marketing, R&D, procurement, manufacturing and distribution. Tasks within this supply chain typically consist of access to one or more data services. For example, approval of some engineering changes might involve updates in data services of a number of component manufacturers. Technically, these tasks can be modeled into distributed transactions. For many of these transactions, it is crucial that global atomicity (among other properties) is guaranteed: either all individual sub-tasks get succeeded or nothing takes effect.

Atomic commit protocols are a key element in supporting global atomicity of distributed transactions. *Two-phase commit protocol (2PC)* is the de facto standard atomic commit protocol [4][6]. It is widely agreed that 2PC is important to guarantee correctness properties in the complex distributed world whilst at the same time it reduces parallelism due to high disk and message overhead and locking during windows of vulnerability [7]. Whilst application semantics and alternative correctness criteria are explored to increase parallelism, global atomicity and 2PC will still be used to simplify application development. There are a number of optimizations to the basic 2PC [5][6][11]. Some of them are

so widely used that they are built into commercial systems and become part of transaction processing standards.

The dynamic and heterogeneous nature of Web services based applications sets specific requirements on applying different optimizations as follows.

1. *Compatibility.* The optimizations might be incompatible with each other and therefore cannot be jointly applied.
2. *Dynamic choices.* No optimization is the best in all run-time situations. Often the choice of the most suitable optimization can only be made dynamically based on the run-time situation and application demand.
3. *Self-adaptation.* Data services are usually autonomous management entities. The next-generation service providers should be able to automatically tune themselves and apply the optimizations in order to maximize the utilization of its own local resource and the satisfaction of its own combined set of clients.

In this paper, we narrow our focus on one particular set of well-known 2PC optimizations based on presumptions of the outcomes of distributed transactions when no information about the transactions is available at the coordinator. These optimizations are coined with the names *presumed nothing (PrN)*, *presumed abort (PrA)* and *presumed commit (PrC)* [6][8][11].

The compatibility issue has been an active research topic in the multidatabase area for many years. The incompatibility of presumptions was first identified in [1][2], where a new protocol called *presumed any (PrAny)* was proposed. PrAny allows participants with different presumptions to be part of a global multidatabase transaction. PrAny, however, does not allow a participant to change presumption during its entire life time and therefore does not meet the dynamic-choices and self-adaptation requirements where a data service should be able to dynamically adapt to the most appropriate presumption.

There have also been efforts on dynamically choosing the (likely) best presumptions for homogeneous database systems, though PrA is generally considered to “win” the game, due to its simplicity and its better combination with the particularly useful optimization for read-only transactions [9][11]. In

system R\* [9], a coordinator can choose between PrA and PrC on a transaction-by-transaction basis, at the start of 2PC. Different presumptions, however, cannot be mixed in one transaction. This, again, conflicts with the self-adaptation requirement. For example, one data service provider may favor PrC in a period due to less forced disk writes of PrC participants in the commit case. A distributed transaction in favor of PrA as a whole (e.g., most of its sub-transactions are read-only) should not force this service provider to use PrA.

In this paper, we construct an improved 2PC (called *DPr – dynamic presumption*) where different presumptions can be dynamically combined in one transaction. The choice of presumption can be made as late as a participant service provider returns from its normal execution, i.e., just before the 2PC starts. Furthermore, the protocol does not introduce extra overhead to the previous 2PC variants in terms of number of messages and log records. Finally, the protocol is easy to understand and realize.

The rest of this paper is organized as follows. The readers unfamiliar with 2PC can find in Appendix an overview of 2PC and its optimizations based on presumptions. Correctness proofs can also be found there. We start with system model and correctness requirements in Section 2. Section 3 addresses the important issues regarding presumptions and when it is safe to discard information about a transaction. Section 4 presents the DPr 2PC protocol in its entirety. Section 5 discusses previous related work. Finally in Section 6 we conclude with a summary of the main contributions.

## 2. System model and correctness requirements

A distributed transaction is started at a *client* process and expanded when services at other processes are invoked as sub-transactions. The only way for processes to share information is via message passing.

A transaction is terminated via 2PC. 2PC is started at the client process (called *coordinator* of 2PC) when all invocations are successfully returned (the return messages are named *WorkDone*). The processes at service providers are *participants* of 2PC.

Processes may fail by stopping, and may re-start when failures are repaired. Messages can be duplicated and can be lost, but they cannot be corrupted. Timeout is used to detect possible remote process failures and message losses.

2PC must guarantee the *atomic commitment properties* [4]:

- AC1: All processes that reach an outcome reach the same one.
- AC2: A process cannot reverse its outcome after it has reached one.

- AC3: The *Commit* outcome can only be reached if all participants voted *Yes*.
- AC4: If there are no failures and all participants voted *Yes*, then the outcome will be *Commit*.
- AC5: Consider any execution containing only failures that the protocol is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach an outcome.

Furthermore, as stated in [1][2], the log garbage collection property must also hold:

- LGC: all processes can eventually forget about transactions and garbage-collect their logs.

The latter is important because this has much to do with the presumptions, the purpose of which is to leave out some acknowledgement messages and the corresponding logging overhead and at the same time still allows the coordinator to draw correct conclusion about the outcome of transactions when *no information* is available about the transactions in question. Under certain presumption, some participants will never acknowledge the outcome message, rendering it impossible for the coordinator to distinguish between the participants that have successfully terminated and those that are still hanging. A naïve solution is that the coordinator never forgets a transaction without receiving acknowledgements from all participants. While this might allow us to guarantee the atomic commitment properties AC1-AC5, it is impractical, because the coordinator will never be able to discard information pertaining to some terminated transactions either from the main memory or from the stable log.

## 3. Inquiries and presumptions

Traditionally all participants and coordinator adopt a single presumption. Now to support dynamic choice of different presumptions, there are two key issues to be addressed for the coordinator to draw correct conclusion about the outcome of a transaction:

1. During what period of time *must* the coordinator keep the information pertaining to a transaction?
2. In the *no-information* case, how can the coordinator be aware of the presumption used by a participant?

We address the first issue in Sections 4.1 and 4.2 and the second one in Section 4.3. Note that PrN also presumes abort. In other words, the extra messages and log records in PrN that PrA leaves out are in practice not necessary. The existence of PrN is mainly of historical reasons. In the discussions that follow, we focus ourselves on PrA and PrC only.

### 3.1. Participant in-doubt windows

**Definition 1** (*Participant in-doubt window*): A participant is within the *in-doubt window* of a transaction if it cannot decide by itself the outcome of the transaction, i.e., without getting it from the coordinator. An in-doubt window is opened when a participant sends out a *Yes* vote; it is closed when the protocol is terminated at the participant. An in-doubt window is *stable* if the events signifying its opening and closing are on stable storage. A 2PC variant is *in-doubt window complete* if all in-doubt windows will eventually be closed.

Comments:

- In the literature, participant in-doubt windows are sometimes also called *windows of vulnerability* or *prepared states* of participants. The concept is independent of the presumption adopted, though its implementations can bear slight differences in whether some log records are forced or no-forced (Figure 1).
- If a participant is not within an in-doubt window of a transaction, it will *never* ask the coordinator about the outcome of the transaction. Either does it appear to the participant that the 2PC has not started, so it can unilaterally abort the transaction; or has the 2PC already completed at the participant.
- An in-doubt window must be stable. As long as a participant enters an in-doubt window, it must remain so until the in-doubt window is explicitly closed despite of subsequent system failures. More specifically, before sending a *Yes* vote, a participant must force-write a log record to stably record the opening of an in-doubt window. The *Prepare* log record serves this purpose, among others. At the termination of 2PC, a participant must write a (forced or no-forced) outcome log record (*Commit* or *Abort*).
- All in-doubt windows must be closed eventually. There are two reasons for this. For one, 2PC must eventually terminate at all participants (AC5). By its definition, this means that in-doubt windows at all participants must eventually be closed. For the other, the log records pertaining to an in-doubt window can only be discarded when the in-doubt window is closed. This is crucial for guaranteeing the LGC property.

### 3.2. Coordinator forced-availability windows

**Definition 2** (*Coordinator forced-availability window*): For a transaction, a coordinator is within the *forced-availability window* of a participant if it cannot use the presumed outcome of the participant in responding to the inquiries of the participant about the outcome of the transaction. A forced-availability window is *stable* if the events signifying its opening and closing are on stable storage. A 2PC variant is

*forced-availability window complete* if all forced-availability windows will eventually be closed.

Comments:

- Coordinator forced-availability windows are closely related with the current state of the coordinator and the presumptions of the participants. Figure 1 illustrates forced-availability windows in different situations. In the figure, double lines indicate messages that will be re-sent if no acknowledgements are received in time.

In the case of PrA, a forced-availability window is opened when the coordinator decides that the outcome be *Commit*; it is closed when the coordinator is acknowledged that all participants have finished committing the transaction (Figure 1a). Before a forced-availability window is open, if no information about the transaction is available (due, for example, to a system crash), the coordinator can safely decide that the outcome of the transaction be *Abort*, which is the same as the presumed outcome. After a forced-availability window is closed, the coordinator knows for sure that it will never be asked about the outcome of the transaction (since all participants have closed their corresponding in-doubt windows). Notice that when the outcome is *Abort*, no forced-availability window is ever opened (Figure 1b).

The case of PrC is somewhat more sophisticated. A forced-availability window is opened as early as the coordinator sends out a *Prepare* message; it is closed either when it decides that the outcome be *Commit* (Figure 1c) or when it knows for sure that all participants have finished aborting the transaction (Figure 1d). Before the coordinator sends out a *Prepare* message, no participant has ever opened an in-doubt window of the transaction and thus there will be no inquiry about the outcome of the transaction. After the decision of *Commit*, the coordinator can safely respond to any inquiry with the presumed outcome *Commit*. On the other hand, if the outcome is *Abort*, the coordinator has to keep the information about the transaction until it knows for sure that all participants have finished aborting the transaction and so no inquiry about its outcome will come.

- A forced-availability window must be stable. As long as a coordinator enters a forced-availability window, it must remain so until the forced-availability window is explicitly closed despite of subsequent system failures.

In the case of PrA, before sending the *Commit* message, the coordinator must force-write a log record to stably record the opening of the forced-availability window. The *Commit* log record serves this purpose. After receiving all acknowledgements from committed participants, it writes a no-forced *CommitEnd* record (Figure 1a).

In the case of PrC, the coordinator must force-write an *Init* log record, in prior to the *Prepare* message, to stably record

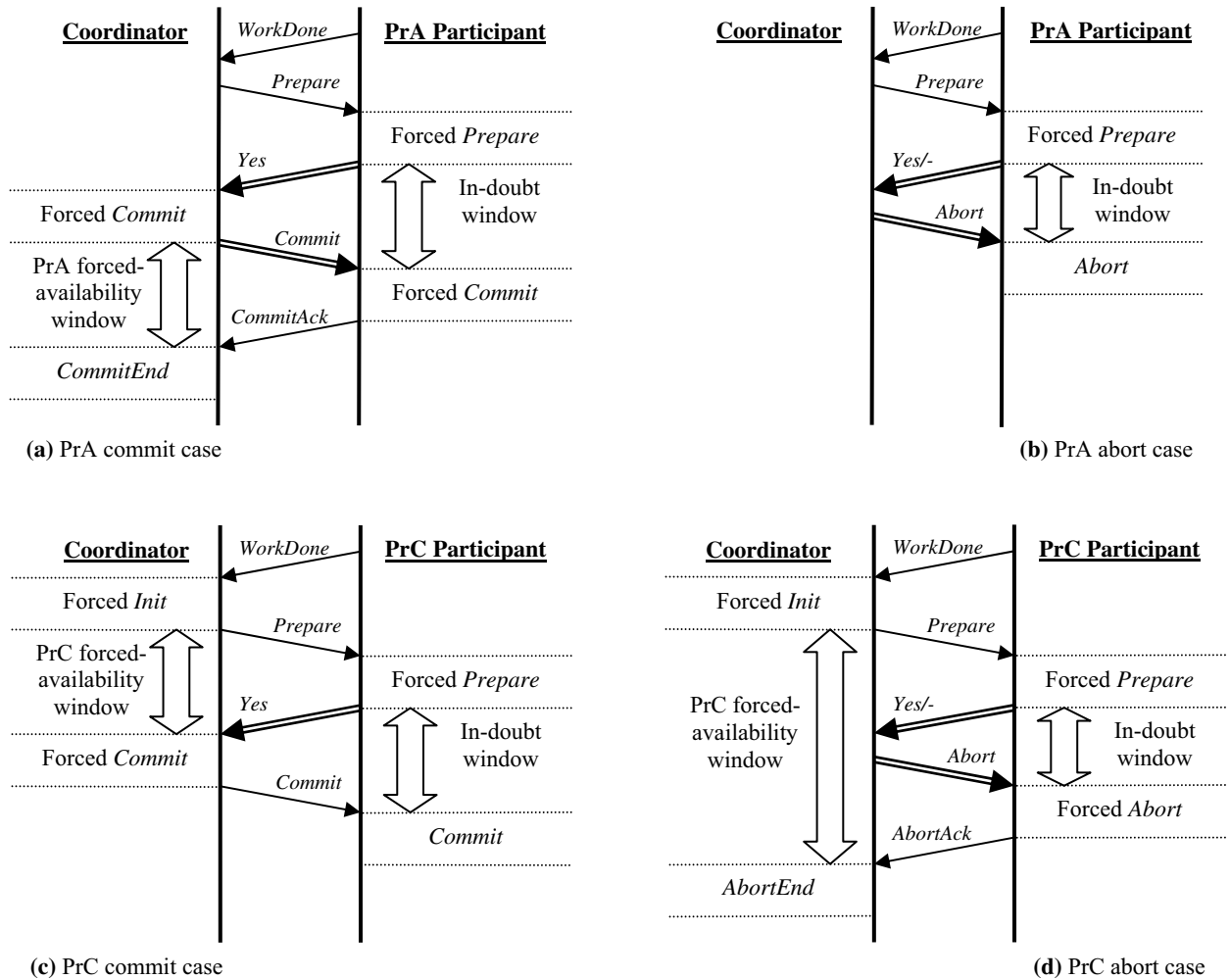


Figure 1. Forced-availability and in-doubt windows

the opening of a forced-availability window. Before sending the *Commit* message, the coordinator force-writes a log record to stably record the closing of the forced-availability window (Figure 1c). The *Commit* log record serves this purpose. (Notice the different roles of the *Commit* log record in PrA and PrC. In PrA, it stabilizes the opening of a forced-availability window, whereas in PrC, it stabilizes the closing of a forced-availability window.) When the coordinator receives acknowledgements from all aborted participants, it writes a no-forced *AbortEnd* log record to record stably the closing of the forced-availability window (Figure 1d).

- All forced-availability windows must be closed eventually. This is crucial for guaranteeing the LGC property, because the log records pertaining to a forced-availability window can only be discarded when the forced-availability window is closed.

### 3.3. Knowing the presumptions

When a participant has finished executing the request from the client, it chooses the presumption to be used and includes this one-bit information (*presumption-bit*) in the return message (*WorkDone*). Later, it also includes the *presumption-bit* in the *Prepare* log record. This stable information is now associated with the in-doubt window and will never be improperly discarded. When the coordinator writes the *Commit* log record, it also includes the presumptions of all participants in the record. This is necessary because the same *Commit* log record serves different purposes for PrA and PrC participants. The explicit knowledge of the presumptions of the participants is needed to properly garbage collect the log: the coordinator must be careful not to discard the *Commit* log record when it closes the forced-availability window for PrC participants while there might still be in-doubt PrA participants.

When a participant inquires about the outcome of the transaction, it includes the presumption-bit in the inquiry message. Usually, a re-sending of *Yes* vote message serves this purpose. If so, the *Yes* vote message must include the presumption-bit. The coordinator will then use this information to give the correct answer.

## 4. The DPr 2PC

### 4.1. Protocol highlights

The basic structure of the DPr 2PC is very much the same as the earlier 2PC variants. Namely, it consists of a voting phase and a decision phase. The types of messages and log records are also the same. We highlight the new features below:

- When a participant has executed a sub-transaction, it chooses a presumption most suitable for the current execution and includes a presumption-bit in the *WorkDone* message to the coordinator.
- The coordinator maintains a forced-availability window for all PrC participants (if any) and a forced-availability window for all PrA participants (if any). Care is taken to properly close all forced-availability windows, so that the log can be correctly garbage collected.
- Each participant includes the presumption-bit in the *Prepare* log record which opens an in-doubt window. Care is taken to properly close all in-doubt windows, so that the log can be correctly garbage collected.
- The participants only acknowledge the outcomes that are different from the presumed ones. The duplicated outcome messages are also acknowledged. (The messages include the presumptions of participants, so the participants know the presumptions even when the in-doubt windows are closed.)
- On recovery from a system crash or on timeout of an expected outcome message, a participant inquires the outcome by re-sending the *Yes* vote message. The message includes the presumption-bit.
- The coordinator responds to the participant with the presumption included in the *Yes* vote message when no information about the transaction is available.

### 4.2. The protocol

A coordinator maintains in main memory a *protocol table* (*PTbl*) that contains an entry for every transaction that has entered 2PC. For every transaction, it maintains a list of participants, their presumptions and votes (if voted).

Before the protocol starts, the coordinator has received return messages from all participants and therefore is aware of their adopted presumptions. The coordinator may choose to override the presumptions chosen by the participants and

include this information in the *Prepare* message. We will not explore this feature further in this paper.

The messages *WorkDone*, *Yes* and log record *Prepare* contain the presumption-bit of the participant. The messages *Prepare* (if the coordinator can override the presumptions), *Commit*, *Abort* and the coordinator log record *Commit* contain presumptions of all participants.

#### 4.2.1. Normal process

##### Voting phase

###### **Coordinator:**

- C0. Insert transaction in *PTbl*. If there are PrC participants, force-writes *Init* log record.
- C1. Send *Prepare* message to all participants and wait for replying vote messages.

###### **Participant:**

- P1. Upon receiving *Prepare* message: If the vote is *Yes*, force-write *Prepare* log record and reply with *Yes* vote message. If the vote is *No*, reply with *No* vote message.

##### Decision phase

###### **Coordinator:**

- C2a. Upon receiving *Yes* vote message:
  - C2a-1. The transaction does not exist in *PTbl*: If the message contains PrC, reply with *Commit* message; otherwise, reply with *Abort* message.
  - C2a-2. The outcome is available in *PTbl*: Reply with that outcome.
  - C2a-3. Otherwise: Update *PTbl*. If all participants have voted *Yes*, force-write *Commit* log record and send *Commit* message to all participants. If there is no PrA participant, remove the transaction from *PTbl*; otherwise wait till *CommitAck* messages from all PrA participants have arrived, then write *CommitEnd* log record and remove the transaction from *PTbl*.
- C2b. Upon receiving *No* vote or timeout of a vote message: Send *Abort* message to all participants whose votes are not *No*. If there is no PrC participant, remove the transaction from *PTbl*; otherwise wait till *AbortAck* messages from all PrC participants have arrived, then write *AbortEnd* log record and remove the transaction from *PTbl*.
- C3a. Upon timeout of *CommitAck* from PrA participant: Re-send *Commit* message to the participant.

C3b. Upon timeout of *AbortAck* from PrC participant: Re-send *Abort* message to the participant.

**Participant:**

P2a. Upon receiving *Commit* message:

P2a-1. First *Commit* message: For PrA participant, force-write *Commit* log record and reply with *CommitAck* message; for PrC participant, write *Commit* log record.

P2a-2. Duplicated *Commit* message (there is already *Commit* log record or the transaction is unknown): If the presumption found in the message is PrA, reply with *CommitAck* message.

P2b. Upon receiving *Abort* message:

P2b-1. First *Abort* message: For PrA participant, write *Abort* log record; for PrC participant, force-write *Abort* log record and reply with *AbortAck* message.

P2b-2. Duplicated *Abort* message (there is already *Abort* log record or the transaction is unknown): If the presumption found in the message is PrC, reply with *AbortAck* message.

P3. Upon timeout of outcome message: Re-send *Yes* vote message.

**4.2.2. Recovery from system failure**

**Coordinator:**

C4. Scan log backward:

For every *Commit* record, if there are associated PrA participants but no corresponding *CommitEnd* record, insert the transaction in *Ptbl*, send *Commit* to all participants and wait for *CommitAck* from all PrA participants.

For every *Init* record without corresponding *Commit* or *AbortEnd* record, insert the transaction in *Ptbl*, send *Abort* to all participants and wait for *AbortAck* from all PrC participants.

Enable normal processing of the protocol.

**Participant:**

P4. Scan log backward:

For every *Prepare* record without corresponding *Commit* or *Abort* record, send *Yes* vote message to coordinator.

Enable normal processing of the protocol.

**4.2.3. Garbage collection of logs**

The log garbage collection program is run periodically. The logs of a transaction are garbage collected if the corresponding windows (forced-availability windows for coordinator and in-doubt windows for participants) are closed.

**Coordinator:**

C5. Scan log backward:

For every *CommitEnd* or *AbortEnd* record, discard all record of the transaction.

For every *Commit* record without corresponding *CommitEnd* record, if there is no associated PrA participant, discard all record of the transaction.

**Participant:**

P5. Scan log backward:

For every *Commit* or *Abort* record, discard all records of the transaction.

**5. Related work**

The issue of selecting an appropriate presumption dynamically or combining different presumptions in one transaction is not new. Previous research efforts have resulted in various solutions in specialized cases. The DPr 2PC can be viewed as generalizing the previous specialized solutions.

In system R\* [9], the coordinator can choose between PrA and PrC on a transaction-by-transaction basis, at the start of 2PC. The coordinator includes the chosen presumption in the *Prepare* message. Every participant includes the presumption in the *Prepare* log record so this information is not lost before the protocol terminates. The presumption is also included in the inquiry messages by the recovery processes of participants, so the coordinator can use this information in responding to inquiries when no information is available in the protocol table. Garbage collection of logs is not discussed.

The presumed-either 2PC [3] is a further optimization of the system R\* 2PC using the principle similar to group commit. Instead of force-writing an *Init* log record as in the PrC 2PC, the coordinator writes a no-forced *Participant* record when some participants of the transaction are known. The *Participant* records might be piggybacked by forced writes possibly of some other transactions. When 2PC starts, if all *Participant* records are already in disk, the coordinator may choose either PrA or PrC; otherwise PrA is the only choice left.

Our protocol reduces to the 2PC in R\*, when all participants adopt the same presumption.

PrAny [1][2] allows a transaction to have participants with different presumptions. DPr shares some similarity with PrAny. Particularly, the concept of forced-availability window

corresponds to “safe state” in [2]. The main difference lies in how the coordinator knows the presumption of a participant. In PrAny, a coordinator records the presumption employed by each participant on stable storage. As a result, a participant is restricted to one particular presumption during its entire lifetime. Moreover, PrAny is not strictly correct with regard to garbage collection of logs, though the information about presumptions that is not garbage collectable is very small in the static world.

Our protocol reduces to PrAny (and satisfies completely the LGC property) when no participant will change its presumption in its entire lifetime.

[1] shows a different approach to dynamically applying presumptions. A participant still cannot change its used presumption. Instead, an agent sits between the coordinator and a participant as an adapter. Additional disk writes are necessary by the agent when the presumption used by the participant is different from the one known to the coordinator. Although this approach may save a few acknowledgement messages in some cases, it in effect increases disk writes at the agent.

With regard to the requirements presented in the Introduction, our protocol supports all three whereas the previous work only partially supports some of them: PrAny supports compatibility; system R\* and presumed-either support dynamic choices; none of these support self-adaptation of presumptions for service providers.

## 6. Conclusion

We observed some new issues concerning supporting distributed transactions in the next-generation applications based on Web services and developed a 2PC protocol addressing these issues for a set of well-known 2PC optimizations. Particularly the protocol allows participants with different presumptions to be dynamically combined in one distributed transaction. This extends the previous work where either all participants adopt the same presumption or participants cannot change presumptions in their entire lifetime. The protocol bears the same structure of the previous 2PC variants, is easy to understand and implement, and does not introduce extra overhead in terms of number of messages and log records.

## Acknowledgements

This research is supported in part by the Norwegian Research Council under the project Arctic Beans during the first author’s sabbatical leave at Georgia Tech.

## 7. References

- [1] Al-Houmaily, Y. J. and P. K. Chrysanthis, “Dealing with Incompatible Presumptions of Commit Protocols in Multidatabase Systems”, In *Proceedings of the 11th ACM Annual Symposium on Applied Computing*, 1999.
- [2] Al-Houmaily, Y. J. and P. K. Chrysanthis, “Atomicity with Incompatible Presumptions”, In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1999.
- [3] Attaluri, G. K. and K. Salem, “The Presumed-Either Two-Phase Commit Protocol”, *IEEE Transactions on Knowledge and Data Engineering*, 14(5), pp 1190-1196, 2002.
- [4] Bernstein, P. A., V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Reading, MA: Addison-Wesley, 1987.
- [5] Chrysanthis, P. K., G. Samaras and Y. J. Al-Houmaily, “Recovery and Performance of Atomic Commit Processing in Distributed Database Systems”, In V. Kumar and M. Hsu (eds.), *Recovery Mechanisms in Database Systems*, pp 370-416, Prentice Hall PTR, 1998.
- [6] Gray, J. and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.
- [7] HPTS’03, High Performance Transaction Systems Workshop, “Debate on Web services and Transactions”, October 14, 2003, <http://research.sun.com/hpts2003/>
- [8] Lamson, B. W. and David B. Lomet, “A New Presumed Commit Optimization for Two Phase Commit”, In *Proceedings of 19th International Conference on Very Large Data Bases*, 1993.
- [9] Mohan, C., Bruce G. Lindsay and Ron Obermarck, “Transaction Management in the R\* Distributed Database Management System”. *ACM Transactions on Database Systems*, 11(4), pp 378-396, 1986.
- [10] Object Management Group, *CORBA Services Specification*, Chapter 10, Transaction Service Specification, December, 1998.
- [11] Samaras, G., K. Britton, A. Citron and C. Mohan, “Two-Phase Commit Optimizations in a Commercial Distributed Environment”, *Distributed and Parallel Databases*, 3(4), pp 325-360, 1995.
- [12] X/Open Company Ltd., *Distributed Transaction Processing: The XA Specification*. Document number XO/CAE/91/300, 1991.

## Appendix

### A. Overview of 2PC and presumption-based optimizations

2PC consists of two phases: the *voting phase* and the *decision phase*. In the voting phase, the coordinator asks the participants to get prepared to commit by sending a *Prepare* message and collects the votes from the participants. In the decision phase, the coordinator decides and notifies the participants of the outcome of the transaction. If it has collected *Yes* votes from all participants, the outcome is *Commit*; otherwise, the outcome is *Abort*, since either some participant has voted *No* or some vote has not been successfully delivered due to some system or communication failure. With a *No* vote, a participant either has aborted the transaction or decides to abort the transaction unilaterally. With a *Yes* vote, a participant declares that it is prepared to commit: it assures that it is able to either commit or abort, even in case of subsequent system failures. The decision, however, must be made by the coordinator. The participant cannot make any decision unilaterally after sending the *Yes* vote.

The resilience of 2PC to system or communication failures is achieved by logging the relevant events of the protocol. The coordinator force-writes a decision record (*Commit* or *Abort*) prior to notifying the outcome to the participants, so the decision, once made, will survive subsequent system failures. Each participant force-writes a *Prepare* record prior to sending the *Yes* vote. The *Prepare* log record stores enough information for the transaction to either commit or abort, even in case of subsequent system failures. Before sending an acknowledgement, the participant force-writes (possibly in a lazy fashion) an outcome record (*Commit* or *Abort*). This assures that the participant will never ask the coordinator about the outcome of the transaction. After receiving acknowledgements from all participants, the coordinator writes an end record (*CommitEnd* or *AbortEnd*) and the protocol terminates.

The coordinator maintains a *protocol table* in main memory. For every transaction, the protocol table records the status information of every participant (vote, acknowledgement etc.). The coordinator uses this information to conduct the progress of the protocol. The information of a transaction is discarded from the protocol table when the protocol terminates. The entire protocol table is lost upon system crash at the coordinator. On recovery, the log is used to re-establish the protocol table for non-terminated transactions at the time of the crash. Note that if no information about a transaction is found in the protocol table, either has the protocol terminated, or the decision has not been made yet before the crash, thus the coordinator can safely decide to abort the transaction. This is the hidden presumption of the *presumed nothing* (PrN) 2PC.

The *presumed abort* (PrA) 2PC makes this presumption explicit to spare a few messages and log records. When the outcome is *Abort*, the coordinator simply notifies the participants of the outcome. No log record is needed and no acknowledgements from participants are expected. At a participant, the *Abort* record is not required to be force-written, since this is the end of the protocol at the participant: no acknowledgement will be sent to the coordinator.

The presumed commit (PrC) 2PC, on the other hand, is designed to reduce the cost when the outcome is *Commit*. Missing information about a transaction in the protocol table is interpreted by the coordinator as the *Commit* outcome of the transaction. To avoid making false interpretation, the coordinator must force-write an *Init* log record before sending the *Prepare* message. When the decision is *Commit*, the coordinator force-writes a *Commit* record and then notifies the participants of the outcome. No acknowledgements are expected from the participants. At the participants, the *Commit* log records are not required to be force-written. Therefore when the outcome is *Commit*, PrC spares the messages for acknowledgements and the *CommitEnd* log record at the coordinator, and the *Commit* records at participants become no-forced. All these savings are at the cost of the extra forced *Init* log record, no matter what the outcome would be.

### B. Correctness

**Theorem 1** (LGC participant): A 2PC participant guarantees the LGC property if and only if it is in-doubt window complete.

Proof:

IF: That a participant is in-doubt window complete indicates that it will eventually close the in-doubt window of any transaction. After that, it will never ask about the transaction. Thus all information about the transaction can be discarded.

ONLY IF: We prove by contradiction. Consider an in-doubt window of a transaction that will never be closed. Suppose the logs pertaining to that transaction can be garbage collected. Now consider the case where the outcome is the same as the presumed outcome of the participant (*Commit* for PrC participant and *Abort* for PrA participant). After sending out an outcome message, the coordinator can forget about the transaction without the acknowledgement of the participant. Now none of the coordinator and participant has any information about the transaction. If the outcome message is lost, the transaction can never be atomically terminated. □

**Theorem 2** (LGC coordinator): A 2PC coordinator guarantees the LGC property if and only if it is forced-availability window complete.

Proof:

IF: That a 2PC is forced-availability window complete implies that for any transaction, the coordinator will eventually close all forced-availability windows of the transaction. When all forced-availability windows are closed, the coordinator can discard all information about the transaction. As long as the coordinator can obtain the presumptions used by the participants (see Section 4.3 on how this can be achieved), the coordinator can always use the presumed outcome in responding to the inquiries in the no-information case.

ONLY IF: We prove by contradiction. Consider the forced-availability window of a transaction that is never closed. Suppose now the logs pertaining to the transaction can be garbage collected. Now the coordinator has no information about the transaction after a system crash. If an inquiry comes, the coordinator will reply using the presumed outcome. This is incorrect, because the forced-availability window is still open and the outcome is the opposite of the presumed one. □

**Theorem 3** (Correctness of DPr 2PC): The DPr 2PC satisfies the properties AC1-5 and LGC.

Proof:

AC3 and AC4 are obvious from code (C2a-3).

AC2 is also straightforward. When a participant reaches an outcome, it closes the in-doubt window and will never inquire about the transaction again (P2a-1 and P2b-1). The coordinator only makes a decision in two disjoint cases (C2a-3 and C2b). Nowhere in the protocol will the coordinator make a new decision.

AC1:

In the protocol where the decision is made by the coordinator, AC1 can also be stated like this: if a decision has been made by the coordinator, all participants will reach the same outcome.

In the *Commit* case, the coordinator force-writes a *Commit* log record before notifying the participants of the decision (C2a-3). This opens a forced-availability window for PrA participants and closes a forced-availability window for PrC participants. If a PrA participant does not receive the *Commit* message, it inquires the coordinator by re-sending a *Yes* vote message (P3). Since the coordinator has not received *CommitAck* from this PrA participant, the forced-availability window is still open, it can find the *Commit* decision in the protocol table and reply with a *Commit* message (C2a-2). If a PrC participant does not receive the *Commit* message, it inquires the coordinator by re-sending a *Yes* vote message (P3). If the coordinator still has the transaction in the protocol table, it can find the *Commit* decision there (C2a-2). If the coordinator does not have any information about the transaction, it will use the presumed outcome of PrC and reply with a *Commit* message (C2a-1). In all these cases, the coordinator will respond with the same *Commit* message. Thus all participants will reach the same *Commit* outcome.

The *Abort* case is similar. If a PrA participant does not receive the *Abort* message, it inquires the coordinator by re-sending a *Yes* vote message (P3). If the coordinator still has the transaction in the protocol table, it can find the *Abort* decision there (C2a-2). If the coordinator does not have any information about the transaction, it will use the presumed outcome of PrA and reply with an *Abort* message (C2a-1). If a PrC participant does not receive the *Abort* message, it inquires the coordinator by re-sending a *Yes* vote message (P3). Since the coordinator has not received *AbortAck* from this PrC participant, the forced-availability window is still open (an *Init* record without *AbortEnd* record), it can find the *Abort* decision in the protocol table and reply with an *Abort* message (C2a-2). In all these cases, the coordinator will respond with the same *Abort* message. Thus all participants will reach the same *Abort* outcome.

AC5:

There are three possible failures: coordinator failure, participant failure and communication failure.

The open forced-availability windows are stably logged and will survive system failures. The coordinator recovery procedure can re-establish all forced-availability windows that are open at the moment of coordinator failures (C4). Similarly, the open in-doubt windows are stably logged and will survive system failures. The participant recovery procedure can re-establish all in-doubt windows that are open at the moment of coordinator failures (P4).

A coordinator will detect possible participant and communication failures by timeout and re-sends an outcome message (C3a and C3b). A participant will detect possible coordinator and communication failure by timeout and re-send a *Yes* vote message (P3).

Thus, if all these possible failures are repaired and no new failures occur for sufficiently long, both the coordinator and the participants will eventually reach an outcome.

LGC:

The protocol is forced-availability window complete (C2a-3 and C2b) and in-doubt window complete (P2a-1 and P2b-1). According to Theorem 1 and Theorem 2, both the coordinator and the participants satisfy LGC. □