# 2-PHASE COMMIT PROTOCOL

Jens Lechtenbörger, University of Münster, Germany

## SYNONYMS

XA standard, distributed commit protocol

## DEFINITION

The *2-phase commit (2PC) protocol* is a distributed algorithm to ensure the consistent termination of a transaction in a distributed environment. Thus, via 2PC an unanimous decision is reached and enforced among multiple participating servers whether to commit or abort a given transaction, thereby guaranteeing atomicity. The protocol proceeds in two phases, namely the prepare (or voting) and the commit (or decision) phase, which explains the protocol's name.

The protocol is executed by a *coordinator* process, while the participating servers are called *participants*. When the transaction's initiator issues a request to commit the transaction, the coordinator starts the first phase of the 2PC protocol by querying—via *prepare messages*—all participants whether to abort or to commit the transaction. If all participants vote to commit then in the second phase the coordinator informs all participants to commit their share of the transaction by sending a *commit message*. Otherwise, the coordinator instructs all participants to abort their share of the transaction by sending an *abort message*. Appropriate log entries are written by coordinator as well as participants to enable restart procedures in case of failures.

## HISTORICAL BACKGROUND

Essentially, the 2PC protocol is modeled after general contract law, where a contract among two or more parties is only established if all parties agree; hence, the underlying idea is well-established in everyday life. According to [3] the first known implementation in a distributed system was performed by Nico Garzado for the Italian social security system in the early 1970s, while the protocol's name arose in the mid 1970s. Early scientific presentations are given by Gray [2] and by Lampson and Sturgis [4]. Since then an API for the 2PC protocol has been standardized under the name XA within the X/Open Distributed Transaction Processing (DTP) model [8], and this API has been incorporated into several middleware specifications and implemented in numerous software components.

## SCIENTIFIC FUNDAMENTALS

The 2PC protocol as described and analyzed in detail in [7] assumes that parts of a single (distributed) transaction involve resources hosted by multiple *resource managers* (e.g., database systems, file systems, messaging systems, persistent programming environments), which reside on possibly different nodes of a network and are called *participants* of the protocol. For every transaction one *coordinator* process, typically running on the node of that participant where the transaction was initiated, assumes responsibility for executing the 2PC protocol; alternative strategies for selecting (and transferring) the coordinator are discussed in [7]. The states through which coordinator and participants move in the course of the protocol are illustrated in Fig. 1 and Fig. 2, resp., and explained in the following. Such statecharts represent finite state automata, where ovals denote states, labeled arcs denote state transactions, and arc labels of the form "precondition/action" indicate that (a) the state transition is only enabled if the precondition is satisfied and (b) the given action is executed when the state is changed.

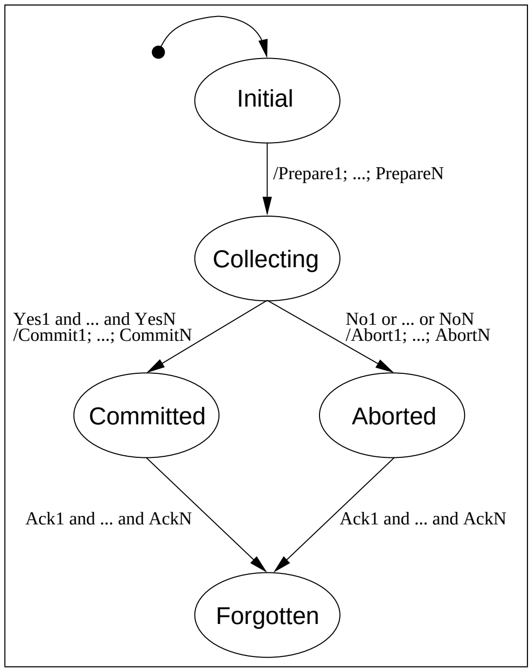

Figure 1: Statechart for coordinator (given $N$ participants)

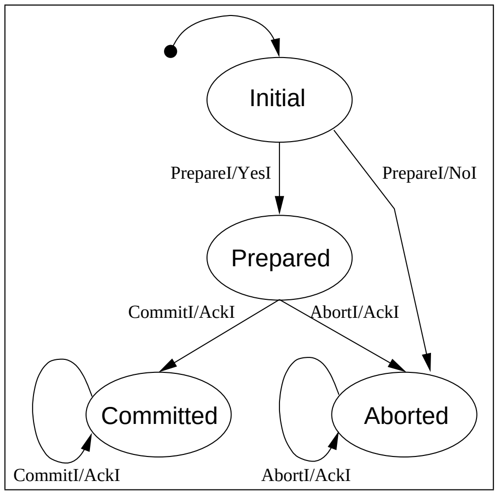

Figure 2: Statechart for participant $I$

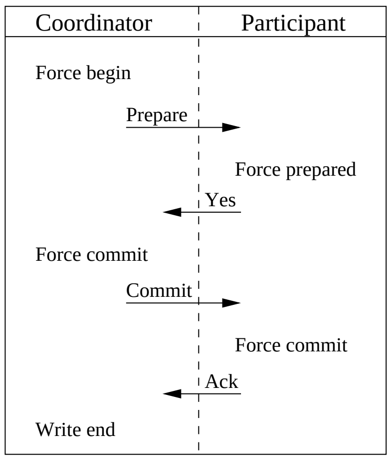

Figure 3: Actions for transaction commit in the basic protocol

**Basic protocol** As long as a transaction is still executing ordinary operations, coordinator as well as all participants operate in the Initial state. When the coordinator is requested to commit the transaction, it initiates the first phase of the 2PC protocol: To capture the state of the protocol's execution (which needs to be available in case of protocol restarts as explained below), the coordinator first forces a *begin log entry*, which includes a transaction identifier as well as a list of the transaction's participants, to a stable log. Afterwards, the coordinator sends a *prepare* message to every participant, enters the Collecting state and waits for replies.
Upon receiving a prepare message, a participant decides whether it is able to commit its share of the transaction. In either case, suitable log entries for later recovery operations as well as a *prepared log entry* indicating the *vote* ("Yes" or "No") are forced to a stable log, before a response message containing the vote is sent back to the coordinator. In case of a No-vote, the participant switches into the Aborted state and immediately aborts the transaction locally. In case of a Yes-vote, the participant moves into the Prepared state. In the latter case the participant is said to be *in doubt* or *blocked* as it has now given up its local autonomy and must await the final decision from the coordinator in the second phase (in particular, locks cannot be released yet).
Once the coordinator has received all participants' response messages it starts the second phase of the 2PC protocol and decides how to complete the global transaction: The result is "Commit" if all participants voted to commit and "Abort" otherwise. The coordinator then forces a *commit* or *abort log entry* to the stable log, sends a message containing the final decision to all participants, and enters the corresponding state (Committed or Aborted).
Upon receipt of the decision message, a participant commits or aborts the local changes of the transaction depending on the coordinator's decision and forces suitable log entries for later recovery as well as a *commit* or *abort log entry* to a stable log. Afterwards, it sends an acknowledgment message to the coordinator and enters the corresponding final state (Committed or Aborted).
Once the coordinator has received all acknowledgment messages it ends the protocol by writing an *end log entry* to a stable log to enable later log truncation and enters the final state, Forgotten. The actions described for the overall process are summarized in Fig. 3 for the case of a transaction commit. (For multiple participants, the actions simply have to be duplicated; in case of abort, at least one of the participants votes "No", which implies that all occurrences of "commit" are replaced with "abort".)

**Protocol restart** The log entries seen so far are used to restart the 2PC protocol after so-called soft crashes of coordinators or participants, i.e., failures like process crashes which lead to a loss of main memory but which leave secondary storage intact. In particular, as participants always force log entries before sending replies, the coordinator never needs to resend messages for which replies have been received. Moreover, log truncation (garbage collection) may occur once all acknowledgment messages have arrived. Finally, every log entry uniquely determines a state, and the last log entry determines the most recent state prior to a failure. Clearly, failures in the final states (Forgotten for the coordinator and Committed or Aborted for a participant) do not require any

action. For the remaining states, restart procedures are as follows:
If the coordinator fails in the Initial or the Collecting state, it simply restarts the protocol in the Initial state. (Coordinators writing received votes into the log could recover differently from the Collecting state.) If it fails in the Committed or in the Aborted state, it re-sends the decision message to all participants, and continues waiting for acknowledgments in the previous state.
If a participant fails in the Initial state it did not yet participate in the 2PC protocol and is free to decide arbitrarily when asked later on. If it fails in the Prepared state it either waits for the coordinator to announce the decision or actively queries the coordinator or other participants for the decision.
In addition to these restart procedures, coordinator and participants also need to be able to recover from message losses. To this end, standard timeout mechanisms are employed: Whenever a message is sent, a timer starts to run. If the timer expires before an appropriate answer is received, the message is simply resent (assuming that either original message or answer are lost; e.g., if the coordinator is missing some votes in the Collecting state, it resends a prepare message to every participant that did not answer in time). Finally, if repeated timeouts occur in the Collecting state the coordinator may decide to abort the transaction globally (as if an "Abort" vote was received), and a participant may unilaterally abort the transaction in the Initial state if no prepare message arrives.

**Hierarchical and flattened 2PC** New participants enter the 2PC protocol whenever they receive requests (e.g., to execute SQL statements) from already existing participants. In such a situation, the new participant can be regarded as child node of the requesting participant, and all such parent-child relationships form a *participant tree* with the transaction's initiator as root node. To execute the 2PC protocol, that tree may either be used directly or flattened as explained in the following.
For the *flattened 2PC*, one node in the participant tree, e.g., the root node, is chosen as coordinator, and this coordinator communicates directly with every participant contained in the tree to execute the basic 2PC protocol as described above. In contrary, in case of the *hierarchical 2PC*, the root node acts as global coordinator, the leaf nodes are ordinary participants, and the inner nodes are participants with respect to their parents as well as sub-coordinators for their children. Thus, when an inner node receives a 2PC message from its parent, the inner node first has to forward the message to its children before it responds on behalf of the *entire* subtree. For example, a prepare message is forwarded down the tree recursively, and an inner node first waits for all votes of its children before it decides, write a log entry, responds with a vote to the parent, and makes a transition to the Prepared (if all children voted to commit) or Aborted state.

**Optimizations** As the 2PC protocol involves costly operations such as sending messages and forcing log entries, several optimizations of the basic protocol have been proposed. In the following the most common variants based on *presumption* are sketched; further details and techniques such as real-only subtree optimization, coordinator transfer, and three-phase commit (3PC) to reduce blocking are presented in [7].
The key idea for presumption based optimizations is to write less log entries and send fewer messages in a systematic way such that in case of a failure the missing information can be compensated for by suitable presumptions concerning the transaction's state. As the basic protocol described above is not based on any presumptions, it is also called *presumed-nothing protocol.* In contrast, in the *presumed-abort protocol*, which aims to optimize the case of aborted transactions, the essential idea is to omit certain information concerning transaction aborts. If that information is needed but absent later on, abort is presumed. In fact, for the presumed-abort protocol

the coordinator's begin and abort log entries are omitted,

- the participants' abort log entries are not forced, and
- participants do not send acknowledgment messages before entering the Aborted state.

The actions required in case of a transaction abort are summarized in Fig. 4, which indicates significant savings when compared with the actions for the basic protocol shown in Fig. 3. In the presumed-abort variant, if a participant fails after receiving the abort decision from the coordinator and restarts without finding a log entry, it queries the coordinator for the decision. As the coordinator does not find the appropriate log entry (which has never been written) it presumes that the transaction should be aborted and informs the participant accordingly, which leads to a globally consistent decision.

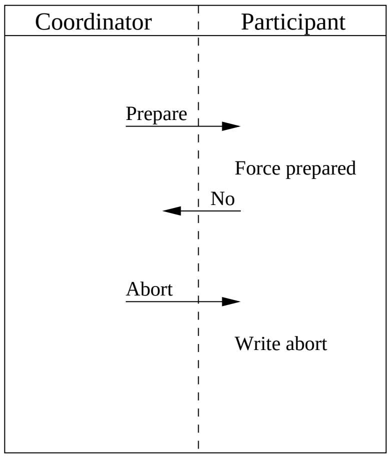

Figure 4: Actions for transaction abort in the presumed-abort variant

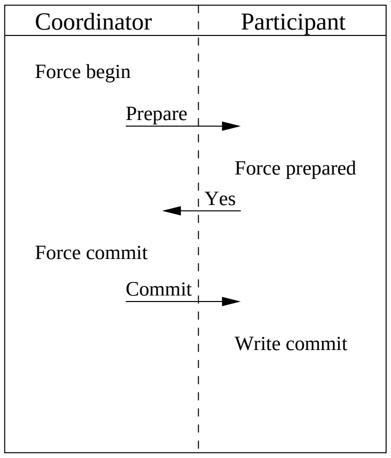

Figure 5: Actions for transaction commit in the presumed-commit variant

Alternatively, in the *presumed-commit protocol*, which aims to optimize the case of committed transactions,
the participants' commit log entries are not forced and

- participants do not send acknowledgment messages before entering the Committed state.

The actions required in case of a transaction commit are summarized in Fig. 5, which again indicates significant savings in comparison to the basic protocol shown in Fig. 3. In this variant, log entries of committed transactions can be garbage collected as missing transactions are presumed to have committed. Thus, if a participant fails after receiving the commit decision from the coordinator and restarts without finding a log entry, it queries the coordinator for the decision. If the coordinator does not find any log entry it presumes that the transaction has committed and informs the participant accordingly, which leads to a globally consistent decision.

## KEY APPLICATIONS

While there is no single key application for the 2PC protocol, it is applicable wherever decentralized data needs to be shared by multiple participants under transactional guarantees, e.g., in e-commerce or e-science settings. More specifically, the 2PC protocol is widely implemented in database systems (commercial as well as open source ones), TP monitors, and message queue systems, where it is used in the background to provide atomicity for distributed transactions. In addition, the XA interface [8] for the protocol, more precisely for the hierarchical presumed-abort variant, has been adopted in the CORBA Transaction Service specified by the OMG [5] and is used as basis for the Java Transaction API (JTA) [6]. Furthermore, the 2PC protocol is also part of the Web Services Atomic Transaction specification [1] to enable the interoperable atomic composition of Web Service invocations.

## CROSS REFERENCE

Transaction, ACID properties, logging and recovery

## RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

[1] Cabrera, L. F., et al. (2005): Web Services Atomic Transaction.
[2] Gray, J. (1978): Notes on Database Operating Systems. In R. Bayer, M.R. Graham, G. Seegmüller (eds.), *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science **60**, Berlin: Springer-Verlag, pp. 393–481.
[3] Gray, J., A. Reuter (1993): *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann.
[4] Lampson, B. W., H. Sturgis: Crash Recovery in Distributed Data Storage Systems. Technical Report, Xerox Palo Alto Research Center, Palo Alto, CA.
[5] OMG (2007): Transaction Service, version 1.4. http://www.omg.org/technology/documents/formal/transaction_service.htm
[6] Sun Microsystems (2007): Java Transaction API (JTA). http://java.sun.com/jta/
[7] Weikum, G., G. Vossen (2002): *Transactional Information Systems — Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, San Francisco, CA.
[8] The Open GROUP (1991): Distributed Transaction Processing: The XA Specification. X/Open Company Ltd, ISBN 1 872630 24 3.