

# CORRECTNESS OF THE TWO-PHASE COMMIT PROTOCOL

Jan van Leeuwen

## Abstract

The 2-phase commit protocol is a standard algorithm for safeguarding the atomicity of transactions in a distributed system. A self-contained description of the 2-phase commit protocol is presented and verified.

Keywords and phrases: distributed systems, client/server model, message passing, atomic actions, two-phase commit protocol, correctness criteria.

## 1 Introduction

In order to deal with the complex issues of communication and control in a distributed system, it is necessary to have a consistent architectural model underlying the design and development of a system. In many distributed systems the client/server paradigm is used to explain the underlying system view, suggesting the possibility of a formal model and correctness proofs of the design. The rationale for the Client/Server (or Customer/Server) model was first described by Gentleman[3], who identified the issues that need to be resolved in any system designed from this perspective. The model is heavily based on the "abstract object" approach to distributed system design (Watson [8]).

The Client/Server model can be viewed as a unifying framework for the high level description and specification of communications and interactions between processes. One or more processes, identified as the clients, request a service from some other process, which acts as the server. The server, after performing the requested service, posts replies to the clients. The client-server relationships exist for the duration of the interaction. For a detailed study of the Client/Server model, see van Leeuwen [7].

The main topic of this note concerns the more involved "atomic" interactions between clients and servers and the need for atomic commit protocols. An integral development and correctness proof is given for a standard 2-phase commit protocol that seems to be appreciably simpler and more clarifying than similar treatments in the current literature (see e.g. [1, 2]).

### 1.1 Managing Atomic Actions

The Client/Server model assumes that clients and servers interact strictly on a request/reply basis. In applications it may be desirable that a client and a server interact in a more complex manner during a session and engage in an activity (a set of operations or "an action") that affects the information stored at the client and the server simultaneously and in an indivisible manner. Activities of this kind are called "transactions" or "atomic

actions". Atomic actions would pose no particular problem if it weren't for the fact that in all realistic applications "exceptions" and "failures" (like link failures or site crashes) can occur during an atomic action. We will use the term "failure" to refer to any abnormal condition that arises during an atomic action. The possibility of failures requires that the effects of an atomic action must be recoverable at all times throughout its elaboration, until the atomic action can be regarded as "safely completed" at both ends. The implementation of atomic actions thus requires two basic facilities (see e.g. Gray [4]):

- (i) a recovery mechanism, i.e., a mechanism for "undo-ing" the effect of one or more atomic actions. Recovery mechanisms are always based on the use of logs that record information on the processing of atomic actions at each site, and on a method for effectuating a rollback. Logs must be recoverable in case of failures and thus must be kept on "stable storage".
- (ii) an atomic commit protocol, i.e., a protocol for detecting "safe completion" and committing the effects of an atomic action at both ends. Atomic commit protocols are atomic actions and thus must be recoverable themselves.

The occurrence of a failure at some site does not necessarily imply that the atomic actions in progress must all be aborted and rolled back. It may be possible for a node to recover to a consistent state, based on information in its log. (This is called "independent recovery".) After recovery, a site may wish to have an atomic action re-started.

## 1.2 Implementation of atomic actions

The implementation of atomic actions in the context of possible failures is a well-studied problem. An excellent introduction to concurrency control and recovery in distributed databases was given by Bernstein et al. [2]. Also, a detailed recommendation for the implementation of atomic commit protocols appears in the CCR standard of ISO [5]. It suggests that atomic commit protocols follow some version of the well-known 2-phase commit protocol due to Gray [4] and Lampson & Sturgis [6], and use the following primitives:

- (i) C - BEGIN
- (ii) C - PREPARE
- (iii) C - READY
- (iv) C - REFUSE
- (v) C - COMMIT
- (vi) C - ROLLBACK
- (vii) C - RESTART

Our main goal will be to give a more refined and complete presentation of the Individual Commit protocol than is usually given (cf. [1, 2]). As it will require no extra effort, we will describe the protocol for the more general case of a client-initiated atomic action that involves multiple servers. It is assumed that the client remains the "coordinator" (or "superior") of the atomic action and thus of the atomic commit protocol. The client

will only initiate the commit protocol (with a C-PREPARE primitive) if it has reason to do so, i.e., if the activity of the atomic action at its own site has ended (which implies that the servers have provided their operation results insofar as needed by the client). At all times during the atomic action, the client and the servers must be ready to honor a C-ROLLBACK or C-RESTART request from any party in the atomic action. Note that the servers only communicate with the client, but not with each other (during the atomic actions).

### 1.3 Atomic commit protocols

Applied to an atomic action, an atomic commit protocol is a distributed algorithm for the client and the servers that should guarantee that they all commit or all abort the atomic action. Following Bernstein et al. [1] the situation for an atomic commit protocol can be rephrased in more precise terms as follows. Each party (client or server) may cast exactly one of two votes: C-READY ("yes") or C-REFUSE ("no"), and can reach exactly one of two decisions: C-COMMIT ("commit") or C-ROLLBACK ("abort"). An atomic commit protocol must satisfy the following requirements:

- AC1 : A party cannot change its vote after it has cast a vote.
- AC2 : All parties that reach a decision reach the same decision.
- AC3 : A party cannot change its decision after it has reached one.
- AC4 : A C-COMMIT decision can be reached only if all parties voted and voted C-READY.
- AC5 : If there are no failures and all parties voted C-READY, then the decision C-COMMIT will be reached by all parties.
- AC6 : Consider any execution of the protocol in the context of permissible failures. At any point in this execution, if all current failures have been repaired and no new failures occur for a sufficiently long period of time, then all parties will eventually reach a decision.

The requirements can be viewed as the minimal correctness criteria for atomic commit protocols. (Except for AC1, the requirements are taken from Bernstein et al. [1].) AC1 through AC5 can usually be satisfied quite easily, but AC6 requires a suitable recovery procedure to be part of the protocol. Note that after voting C-READY, a party (client or server) can not be certain of what the decision will be until it has received sufficient information to decide. Until that moment, we say that the party is "uncertain". If a failure occurs that cuts an uncertain party off, then this party is said to be blocked. A blocked party cannot reach a decision until after the connection to the other parties has been restored. Blocking is usually concluded if no messages arrive during a certain timeout interval within the uncertainty period. (Heuristic commit protocols allow a blocked party to make a calculated guess of the decision. For example, a blocked client may be allowed to commit.) A different situation arises when a party fails (crashes) during its uncertainty period. In this case a more involved recovery procedure may have to be followed (see Bernstein et al. [1] or below).

## 2 The (2-phase) individual commit protocol

The standard (2-phase) atomic commit protocol is as follows, in terms of the recommended CCR primitives. (Note that the desired steps of the recovery procedure after failure are part of the overall protocol, but these are not included in the basic specification below.)

### Individual Commit Protocol

#### Client's Commit

c-0. Vote ready;

#### Phase 1

c-1. **Write** "prepare" record to the Log;

c-2. **Send** C-PREPARE messages to all servers; **activate** timer;

c-3. {Await answer messages (C-READY or C-REFUSE) from all servers using a timer and act as follows}

**Case condition of**

c-3.1. Timeout or C-REFUSE message received:

**begin**

**Write** "rollback" record to the Log;

**Send** C-ROLLBACK messages to all servers;

C-ROLLBACK

**end;**

c-3.2. All servers answered and answered C-READY:

continue with Phase 2

**end;**

{End of Phase 1}

#### Phase 2

c-4. **Write** "commit" record to the Log;

c-5. **Send** C-COMMIT messages to all servers ; **activate** timer;

c-6. {Await answer messages (ACK) from all servers using a timer and act as follows}

**CASE** condition of

c-6.1. Timeout: **Write** "incomplete" record to the Log;

c-6.2. All servers answered (ACK):

**Write** "complete" record to the Log;

**end;**

c-7. C-COMMIT;

{End of Phase 2}

#### Server's Commit

##### Phase 1

s-1. **Await** a C-PREPARE message from the client using timer;  
{The server will only pass this point if it has indeed received a C-PREPARE message from the client or timed out}

s-2. **If** not timed out **then** Vote ready or refuse;

s-3. **Case** condition of

```

s-3.1. ready:
    begin
        Write "ready" record to the Log;
        Send a C-READY message to the client;
        continue with Phase 2
    end;
s-3.2. refuse:
    begin
        Write "refuse" record to the Log;
        Send a C-REFUSE message to the client
    end
s-3.3. Timeout:
        Write "refuse" record to the Log
    end;
{End of Phase 1}
Phase 2
s-4. Await a decision message (C-COMMIT or C-ROLLBACK) from the client
    using timer;
    {The server will only pass this point if it has indeed received a decision
    message from the client or times out}
s-5. Case condition of
s-5.1. a C-COMMIT message was received:
    begin
        Write "commit" record to the Log;
        Send ACK message to the client;
        C-COMMIT
    end;
s-5.2. a C-ROLLBACK message was received:
    begin
        Write "rollback" message to the Log;
        C-ROLLBACK
    end;
s-5.3. Timeout:
        take whatever action to deal with blocking
    end;
{End of Phase 2}

```

Although it is not explicitly specified, each party can unilaterally decide for a C-ROLLBACK at any time if it has not (yet) voted "ready". We assume that the Client's Commit protocol is only initiated by the client after it has voted "ready". If a client never votes or votes "refuse", then it never sends a C-PREPARE message and the servers automatically time out eventually in their protocol. (We could have treated the client as a server in the protocol, but have chosen not to do so in order to save messages.) We require that C-COMMIT messages are ack'ed, but this can be omitted from the protocol without any harm. ( We do not require that C-ROLLBACK messages are ack'ed, but could incorporate it easily if desired.) Voting essentially amounts to determining whether the local activities

in an atomic action have ended and properly been logged or not, but we only need it as an “abstract” operation. Where ever a “Log” is specified in the protocol, the local (client or server) log is meant. A “Phase 2” is not entered automatically after a “Phase 1”, but only on an explicit “continue”.

## 2.1 Correctness proof

It should be an interesting research topic to develop a completely formal “correctness proof” for the Individual Commit protocol. We outline a less formal proof here. We refer to line-numbers as c-0, c-1, etcetera.

**Theorem A** If no timeouts and no failures occur, then the Individual Commit protocol is correct.

**Proof.**

The requirements AC1 through AC5 are trivially satisfied. For AC2, note that if any party (client or server) decides spontaneously for C-ROLLBACK when it can, then all parties must follow suit eventually and cannot decide for anything else. AC6 is vacuously true. □

The next step is to consider the possibility of timeouts and a limited type of failures, namely “loss of protocol messages”. In all cases except one, message loss necessarily leads to a timeout and thus it is sufficient to consider the latter only. The one exceptional case can arise in c-3, when some messages (including a C-REFUSE) have arrived but some have not and the clients acts on the C-REFUSE. In this case the client acts just like it would have in the case of timeout (line c-3.1.). Note that heavily delayed messages are considered “lost”.

**Theorem B** If timeouts and loss of messages can occur but no server gets blocked, then the Individual Commit protocol is correct.

**Proof.**

The requirements AC1, AC3 and AC4 are trivially satisfied. For AC2, observe the following. If a server times out on s-1, it will never send a message and can only decide C-ROLLBACK (if it ever decides, cf. s-3.3.). The client necessarily executes c-3.1. and decides for C-ROLLBACK too. Other servers either time out on s-1 as well or receive the C-ROLLBACK decision in s-4. By assumption no server times out on s-4 (the blocked case). If the client times out on c-3.1., then it decides C-ROLLBACK and the servers can only reach the same decision by the very same argument. If the client times out on c-6.1., the only possible decision in the system is C-COMMIT and all servers must be in Phase 2. As we assume no blocking, all servers will eventually decide C-COMMIT. Thus AC2 is satisfied and, by the latter argument, AC5 as well. AC6 is vacuously true. □

In order to get any further we must some how deal with the problem of blocking. A blocked server timed out on s-5.3. and thus knows that it is blocked, but it is uncertain of the decision that may have been reached. In order to keep the Individual Commit protocol correct, a Server’s Commit Termination Protocol must be added. The purpose of the Server’s Commit Termination protocol is to enable a blocked server to determine the (apparent) decision reached in the system. Several possible strategies for a successful Server’s Commit Termination protocol have been proposed, all based on polling. (For ex-

ample, if another server can be reached and it appears to have timed out on s-1, then the blocked server can decide C-ROLLBACK.) But no Server's Commit Termination protocol can guarantee that it will remove the possibility of blocking. (For example, if a blocked node is cut off permanently, it will forever remain uncertain.) We assume that any server can eventually reach the client again and thus a simple polling of the client will do as a Server's Commit Termination protocol (cf. requirement AC6). We conclude the following result.

**Theorem C** If timeouts and loss of messages can occur, then the Individual Commit protocol enhanced with the Server's Commit Termination protocol is correct.

The final step is to allow timeouts and arbitrary failures, i.e., "loss of protocol messages" and "site crashes". If a site crashes permanently, the Individual Commit protocol will simply continue in the remaining sites and perform as if the messages of the crashed site are lost from some point onwards. The Server's Commit Termination protocol should be extended in this case and somehow detect permanent crashes of the client. In general there is no guaranteed solution that avoids blocking, if permanent crashes can occur. Thus we assume that each site that crashes eventually recovers and resumes the commit protocol. We also assume that a site can actually recover to the point where it crashed, using the information in its log. The only problem now is to determine how to continue with the commit protocol, knowing that the other sites may have advanced in it after the crash. We present a possible recovery protocol below.

#### **Commit Recovery Protocol**

##### **Client's Commit Recovery**

```

cr-1. If the client crashed before c-4 then
    begin
        Write "rollback" record to the Log;
        Send C-ROLLBACK messages to all servers;
    end;
cr-2. If the client crashed after c-4 but before c-6.2. then
    begin
        Write "commit" record to the Log;
        Send C-COMMIT messages to all servers; activate timer;
        {Await answer messages from all servers using timer and act as follows }
        Case condition of
            Timeout:           Write "incomplete" record to the Log;
            All servers answered: Write "complete" record to the Log
        end;
        C-COMMIT
    end;
cr-3. If the client crashed after c-6.2. then
    begin
        perform c-7 if necessary
    end;

```

**Server's Commit Recovery**

- sr-1. **If** the server crashed before s-4 and without having executed s-3.1.  
**then**  
    **begin**  
        C-ROLLBACK  
    **end;**
- sr-2. **If** the server crashed after s-4 while being uncertain  
**then**  
    **begin**  
        use the Server's Commit Termination protocol to remove blocking  
    **end;**
- sr-3. **If** the server crashed after s-4 while being certain  
**then**  
    **begin**  
        perform remaining activity if necessary in the C-COMMIT or  
        C-ROLLBACK  
        (whatever applies)  
    **end;**

We assume that **Write/Send** commands in the Individual Commit protocol are atomic, and thus no crash occurs "in between" a **Write** and the subsequent **Send**. (While the assumption is reasonable, it would nevertheless be of interest to analyse the protocol if this assumption is not made.)

**Theorem D** The Individual Commit Protocol enhanced with the Server's Commit Termination protocol and the Commit Recovery Protocol is a correct atomic commit protocol.  
**Proof.**

If the client crashes before c-4, then each server either has C-ROLLBACK as the only option is uncertain. Thus cr-1 is a correct recovery action, in the sense of satisfying the requirements. If the client crashes in its Phase 2, then each server has either decided C-COMMIT or is uncertain. Replaying Phase 2 (in so far as it its Phase 1 and without sending a C-READY message, then the remaining sites will have progressed on the assumption that its messages are C-REFUSE or "lost". It means that the remaining sites are on their way to decide C-ROLLBACK, and sr-1 is fully consistent with this. If a server crashed after having sent a C-READY message, then either it had decided (and the decision is recovered) or is uncertain. Thus sr-2 and sr-3 are the actions to take. With the earlier analyses it easily follows that the complete protocol satisfies AC1 through AC6 and hence is correct. □

Utrecht, March 1989.



## References

- [1] Bernstein, P.A., V. Hadzilacos and N. Goodman, *Concurrency control and recovery in database systems*, Addison-Wesley Publ. Comp., Reading, Mass., 1987.
- [2] Ceri, S., and G. Pelagatti, *Distributed databases - principles and systems*, McGraw-Hill Book Comp., New York, NY, 1984.
- [3] Gentleman, W.M., *Message passing between sequential processes: the reply primitive and the administrator concept*, Software - P & E 11(1981) 435-466.
- [4] Gray, J., *Notes on data base operating systems*, Report RJ2188, IBM Research Lab., San Jose, Ca., 1978.
- [5] ISO, *Specification of protocols for application service elements - commitment, concurrency and recovery*, draft international standard, ISO TC97/DIS9805.2, 1985.
- [6] Lampson, B., and H. Sturgis, *Crash recovery in a distributed data storage system*, Techn. Rep., Computer Science lab., Xerox - PARC, Palo Alto, Ca., 1976.
- [7] van Leeuwen, J., *The Client/Server model in distributed computing*, Techn. Rep. RUU-CS-88-9, Dept. of Computer Science, University of Utrecht, Utrecht, 1988.
- [8] Watson, R.W., *Distributed system architecture model*, in: B.W. Lampson et al., *Distributed systems - architecture and implementation*, Lect. Notes in Comput. Sci., vol. 105, Springer Verlag, Berlin, 1981, pp.10-43.