A TERMINATION PROTOCOL FOR SIMPLE NETWORK PARTITIONING IN DISTRIBUTED DATABASE SYSTEMS[1]

Ching-Liang Huang and Victor O.K. Li
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-0272

ABSTRACT

Resilient commit protocols for multisite simple network partitioning are studied in this paper. The necessity of termination protocols to make commit protocols resilient in multisite simple network partitioning is presented. A termination protocol that makes the three-phase commit protocol resilient is designed. This protocol is valid even for transient network partitioning. The method can be generalized to design termination protocols for other commit protocols in multisite simple network partitioning.

1. Introduction

A distributed database system enjoys the potential advantage of providing higher availability and reliability. However, this advantage cannot be achieved unless database consistency is guaranteed in the event of failures. Several commit protocols have been proposed [1, 2, 4, 5, 6, 7] to achieve transaction atomicity in the event of site failures for any number of participating sites, and network partitioning for two participating sites. But none of them are resilient in multisite network partitioning, in which the number of participating sites is more than two.

A termination protocol is a protocol that is invoked to consistently terminate transactions when failures occur and render the continued execution of a commit protocol impossible. In this paper, the necessity of termination protocols to make commit protocols resilient in multisite simple network partitioning is presented. A termination protocol that makes the three-phase commit protocol resilient is designed and the proof of its correctness is given. The protocol is valid even for transient network partitioning. This method may be generalized to design termination protocols for other commit protocols in multisite simple network partitioning.

[1]This research is supported in part by the Joint Services Electronics Program under Contract No. F49620-85-C-0071, and in part by the National Science Foundation under Grant No. DCI8519101.

2. Background

A distributed database system supports a database physically distributed over multiple sites interconnected by a computer network. In such a system, there may be more than one site participating in a transaction. Since, by definition, a transaction is a logically atomic operation, to maintain database correctness, transaction atomicity must be enforced. That is, for any given transaction, either the transaction is processed to completion at all participating sites or it appears to have never been executed at all.

Preserving transaction atomicity in the single site case is a well understood problem [1, 3]. When the site has finished processing the transaction but has not yet updated the database, it will decide to commit or abort the transaction (possibly by asking the user). If a commit decision is made, a commit log which contains the current state of the transaction (e.g. the update information) will be stored in stable storage, and then the site will start committing the transaction. If failures occur at any time before the commit log is stored, then immediately upon recovery the site will abort the transaction. If failures occur after the commit log is stored but before the updates are finished, all the updates will be applied again when the site recovers. Because update operations are idempotent (i.e. performing them several times is equivalent to performing them once), the above scheme ensures the atomicity of the transaction.

In the multiple site case, the problem of guaranteeing transaction atomicity is much more difficult because of arbitrary site failures and partitioning of the computer network. To ensure transaction atomicity in this case, each participating site performs either all or none of the updates locally. In addition, all the sites should make the same decision with respect to committing or aborting the transaction.

Protocols for preserving transaction atomicity are called commit protocols. Several commit protocols have been proposed [1, 2, 4, 5, 6, 7]. The two phase commit protocol [1, 2] is the simplest one (Fig. 1). It is a centralized protocol with a single master and with the remaining participating sites acting as slaves. A participating site can be in one of the following four states

: the initial state (q), the wait state (w), the commit state (c) and the abort state (a). The first phase of the protocol begins when the master receives the transaction ("request") and forwards it to the slaves ("Xact"). When a slave receives the transaction, it will partially execute the transaction and send its intent to commit ("yes") or unilaterally abort ("no") the transaction. The second phase begins when the master receives all the responses from the slaves. If all the participating sites agree to commit the transaction, then the master will send out commit commands ("commit") to the slaves, else it will send out abort commands ("abort"). When a slave receives the command, it will act accordingly.

The two phase commit protocol is simple. However, if the master fails and all the operating slaves are in the wait state, the slaves cannot make any decision to commit or abort the transaction (because the master can be in either the commit state or the abort state) and therefore must block the transaction until the failures recover. Even though blocking preserves database consistency, it is highly undesirable because the locks acquired by the blocked transaction cannot be relinquished, rendering those data inaccessible to other transactions. In [4], nonblocking commit protocols were studied, and a three-phase commit protocol and a termination protocol were presented which are nonblocking under site failures. Network partitioning was studied in [7], and some definitions and results from that paper are reviewed next.

A formal model is used to describe commit protocols. Transaction execution at each site is modelled as a finite state automaton (FSA), with the network serving as a common input/output tape to all sites. The states of the FSA for site i are called the local states of site i. The global state of a distributed transaction consists of (1) a global state vector containing the local states of the participating sites, (2) the outstanding messages in the network. During a global state transition, there is exactly one local state transition, which involves a site reading a nonempty string of messages addressed to it, writing a string of messages, and moving to the next local state.

A commit protocol is resilient to a class of failures only if the protocol enforces transaction atomicity and is nonblocking for any failure within that class.

Network partitioning is divided into two classes:

1. simple partitioning in which the sites are partitioned into exactly two sets with no communication between the sets.

2. multiple partitioning in which the sites are partitioned into more than two sets.

Two models are considered for network partitioning:

1. pessimistic model where all messages are lost at the time partitioning occurs.

2. optimistic model where no messages are lost at the time partitioning occurs ; instead, undeliverable messages are returned to the sender.

Some results on the existence of resilient commit protocols for network partitioning have been developed.

Theorem: There exists no protocol resilient to a network partitioning when messages are lost.

Theorem: There exists no protocol resilient to a multiple network partitioning.

For two-site simple partitioning with return of messages (optimistic model), some results have been derived.

Definition: Concurrency Set C(s) Let s be an arbitrary local state of a commit protocol P. The concurrency set of s is the set of all local states that are potentially concurrent with it in the execution of P.

Definition: Sender Set S(s) Let s be an arbitrary local state of a commit protocol P, and let M be the set of messages that can be received by s in the execution of P. $S(s) = \{ t \mid t \text{ sends } m, m \in M \}$

Two rules are defined for designing resilient commit protocols for two-site simple partitioning with return of messages.

Rule(a): For a state s_i: if its concurrency set, $C(s_i)$, contains a commit state, then assign a timeout transition from s_i to a commit state; else assign a timeout transition from s_i to an abort state.

Rule(b): For state s_j: if t_i is in $S(s_j)$ and t_i has a timeout transition to a commit (abort) state, then assign an undeliverable message transition from s_j to a commit (abort) state, upon the receipt of an undeliverable message.

Using the above two rules, an extended two-phase commit protocol augmented with timeout transitions and undeliverable message transitions is derived (Fig. 2). It has been proved that these two rules are necessary and sufficient for making protocols resilient to two-site simple partitioning with return of undeliverable messages [7].

3. Motivation

The extended two-phase commit protocol shown above works correctly in optimistic two-site simple partitioning; however, we note that it does not work in the multisite case by the following observation:

Consider a transaction with three participating sites where $site_1$ is the master. Suppose a partitioning occurs

with a global state vector of $<p_1,w_2,w_3>$ and outstanding messages $< - ,commit_2,commit_3>$, i.e. the master is in the prepare state, the slaves are in the wait state and the master has sent out commit messages. If the partitioning causes $site_3$ to be separated from $site_1$ and $site_2$, and $commit_3$ undeliverable, then $site_2$ will receive $commit_2$ and commit while $site_3$ will make a timeout transition and abort.

The extended two-phase commit protocol does not work in the multisite case due to the following facts:

1. The wait state (w) of a slave contains both a commit and an abort in its concurrency set.

2. There exists a local state (the wait state of a slave) that is noncommittable and contains a commit in its concurrency set.

(A local state is called committable if occupancy of that state by any site implies that all sites have voted yes on committing the transaction. Otherwise, it is called noncommittable [4].)

From the above observations, we get the following lemmas.

Lemma 1: A commit protocol P can be made resilient to optimistic multisite simple network partitioning only if there does not exist a local state s in P, such that the concurrency set of s contains both a commit and an abort state.

Proof: Assume there exists a local state s of $site_i$ that contains both a commit and an abort in its concurrency set C(s). If $site_i$ is separated from the other participating sites by the network partitioning when it is in state s and it has not obtained any information about whether the other sites commit or abort the transaction, then as long as the network is partitioned, $site_i$ cannot tell whether the other sites commit or abort the transaction. Therefore $site_i$ must be blocked; else no matter whether $site_i$ commits or aborts the transaction before the network recovers, the database would probably become inconsistent.

Q.E.D.

Lemma 2: A commit protocol P can be made resilient to optimistic multisite simple network partitioning only if there does not exist a local state s, such that s is noncommittable and the concurrency set of s contains a commit state.

Proof: Assume there exists a local state of $site_i$ that is noncommittable and contains a commit in its concurrency set. If $site_i$ is separated from the other participating sites by the network partitioning when it is in state s and all messages addressed to $site_i$ become undeliverable, then $site_i$ should not commit because it cannot ensure that all the other participating sites agree to commit. However, $site_i$ should not abort either because some participating site may have committed already. Therefore, site i must be blocked.

Q.E.D.

The two lemmas above is similar to the Fundamental Nonblocking Theorem in [4] which considers site failures instead of network partitioning.

The three-phase commit protocol [4] (Fig. 3) satisfies both Lemma1 and Lemma2; however, we found that augmenting this protocol by timeout transitions and undeliverable message transitions using Rule(a) and Rule(b) shown in Sec. 2 does not make it resilient to optimistic multisite simple network partitioning by the following observation:

Consider a transaction of three participating sites where $site_2$ and $site_3$ are slaves, and abort ϵ $C(w_3)$, commit ϵ $C(p_2)$, p_2 ϵ $C(w_3)$. By Rule(a), the timeout transition from w_3 should go to the abort state and the timeout transition from p_2 should go to the commit state. If $site_3$ is in state w_3 waiting for $prepare_3$ and $site_2$ is in state p_2 waiting for $commit_2$ when partitioning occurs which renders $prepare_3$ undeliverable, then $site_3$ will timeout and abort while $site_2$ will timeout and commit. Therefore, the database becomes inconsistent.

4. Necessity of termination protocol

The three-phase commit protocol is the simplest commit protocol that satisfies both Lemma1 and Lemma2. However, it cannot be made resilient to optimistic multisite simple network partitioning by simply augmenting it with timeout and undeliverable message transitions. If we add some more phases to our commit protocol, will timeout and undeliverable message transitions be sufficient to make the protocol resilient ? The following lemma yields a negative answer.

Lemma 3: If a commit protocol is not resilient to optimistic multisite simple network partitioning, then timeout transitions and undeliverable message transitions are not sufficient for making it resilient.

Proof: It has been proved that protocols in which each state transition reads at most one message are equivalent in power to more general protocols that read an arbitrary number of messages per state transition [6]. Therefore, we only consider protocols of the first type in this proof.

Assume network partitioning will cause all outstanding messages transmitted in the network at the time of partitioning between pairs of sites located in different partitions to be returned to the senders, i.e. Let m_{ij} be the message sent by $site_i$ to $site_j$. If $site_i$ and $site_j$ are located in different partitions and m_{ij} are outstanding when partitioning occurs, then m_{ij} will be returned to $site_i$.

Suppose the network is divided into two partitions:

G_1 and G_2. We can always name the participating sites of a transaction such that all the sites in G_1 precede all the sites in G_2. i.e. $site_i$ is in G_1 for $1 \leq i \leq k$, $site_j$ is in G_2 for $k+1 \leq j \leq n$ where n is the number of participating sites, $n > 2$, and $1 \leq k < n$.

Assume there exists a commit protocol P that is not resilient to optimistic multisite simple network partitioning and can be made resilient by augmenting it with only timeout and undeliverable message transitions. Let $H^0 H^1 \ldots H^c$ be the global state sequence of a failure-free execution of P that commits a transaction.

Let $H^i = (S^i, M^i)$ where the global state vector $S^i=(s_1^i, \ldots, s_n^i)$, and the outstanding messages:

$$M^i = \begin{matrix} m_{1,1}^i & \ldots & m_{1,k}^i & m_{1,k+1}^i & \ldots & m_{1,n}^i \\ \vdots & & \vdots & \vdots & & \vdots \\ m_{k,1}^i & \ldots & m_{k,k}^i & m_{k,k+1}^i & \ldots & m_{k,n}^i \\ m_{k+1,1}^i & \ldots & m_{k+1,k}^i & m_{k+1,k+1}^i & \ldots & m_{k+1,n}^i \\ \vdots & & \vdots & \vdots & & \vdots \\ m_{n,1}^i & \ldots & m_{n,k}^i & m_{n,k+1}^i & \ldots & m_{n,n}^i \end{matrix}$$

$m_{p,q}^i$ is the set of outstanding messages in the network sent by $site_p$ to $site_q$ when the global state is H^i.

Let $UD(m_{p,q}^i)$ denote the set of undeliverable messages of $m_{p,q}^i$, $\underline{H}^i = (S^i, \underline{M}^i)$ be the global state resulting from a partitioning that occurs during global state H^i.

$$\underline{M}^i = \begin{matrix} X_{1,1}^i & \ldots & m_{1,k}^i & \phi & \ldots & \phi \\ \vdots & & \vdots & \vdots & & \vdots \\ m_{k,1}^i & \ldots & X_{k,k}^i & \phi & \ldots & \phi \\ \phi & \ldots & \phi & X_{k+1,k+1}^i & \ldots & m_{k+1,n}^i \\ \vdots & & \vdots & \vdots & & \vdots \\ \phi & \ldots & \phi & m_{n,k+1}^i & \ldots & X_{n,n}^i \end{matrix}$$

where $X_{f,f}^i = m_{f,f}^i \cup UD(m_{f,k+1}^i) \cup \ldots \cup UD(m_{f,n}^i)$

for f = 1 to k

$m_{f,f}^i \cup UD(m_{f,1}^i) \cup \ldots \cup UD(m_{f,k}^i)$

for f = k+1 to n

Let $\underline{M}_j^i = COLUMN_j(\underline{M}^i)$.

Suppose F_j is the transition function that moves $site_j$ to a final state during partitioning, i.e. $F_j(s_j^i, \underline{M}_j^i)$ is the resulting final state of $site_j$ when partitioning occurs during H^i.

Let r be the smallest i such that a simple partitioning which occurs while the transaction is in H^i results in the transaction being committed, i.e. $F_j(s_j^r, \underline{M}_j^r) =$ commit for j = 1 to n. By the choice of r, we have $F_j(s_j^{r-1}, \underline{M}_j^{r-1}) =$ abort for j=1 to n for the same partitioning.

Let the global transition from H^{r-1} to H^r be made by $site_p$ reading a message u_{qp} sent by $site_q$ and sending some (null or nonnull) messages $\{ u_{p1} \ldots u_{pn} \}$. Therefore,

$$M^r = M^{r-1} - \begin{matrix} \phi & \ldots & \phi \\ \vdots & & \vdots \\ \phi & u_{qp} & \phi \\ \vdots & & \vdots \\ \phi & \ldots & \phi \end{matrix} + \begin{matrix} \phi & \ldots & \phi \\ \vdots & & \vdots \\ u_{p1} & \ldots & u_{pn} \\ \vdots & & \vdots \\ \phi & \ldots & \phi \end{matrix}$$

We consider the following four cases :

(1) $1 \leq q \leq k$, $1 \leq p \leq k$

then $\underline{M}_j^r = \underline{M}_j^{r-1}$ for $k+1 \leq j \leq n$

(2) $1 \leq q \leq k$, $k+1 \leq p \leq n$

then $\underline{M}_j^r = \underline{M}_j^{r-1}$ for $1 \leq j \leq k$, $j \neq q$

(3) $k+1 \leq q \leq n$, $1 \leq p \leq k$

then $\underline{M}_j^r = \underline{M}_j^{r-1}$ for $k+1 \leq j \leq n$, $j \neq q$

(4) $k+1 \leq q \leq n$, $k+1 \leq p \leq n$

then $\underline{M}_j^r = \underline{M}_j^{r-1}$ for $1 \leq j \leq k$

Since only $site_p$ makes a transition during the global transition from H_{r-1} to H_r, $s_j^r = s_j^{r-1}$ for $j \neq p$. So in all cases, there exists a $site_j$ such that $s_j^r = s_j^{r-1}$ and $\underline{M}_j^r = \underline{M}_j^{r-1}$. Hence, $F_j(s_j^{r-1}, \underline{M}_j^{r-1}) = F_j(s_j^r, \underline{M}_j^r)$ which is a contradiction.

Therefore, there does not exist such a protocol P.

Q.E.D.

5. A termination protocol for three-phase commit in multisite simple network partitioning

Since timeout transitions and undeliverable message transitions are not sufficient to make commit protocols resilient to multisite simple partitioning, we need to invoke another protocol to consistently terminate the transactions when partitioning occurs. This protocol is called a termination protocol. In this section, we present a termination protocol that makes the three-phase commit protocol resilient to optimistic multisite simple network partitioning.

5.1. Assumptions

We make several assumptions for the following discussion:

1. all undeliverable messages due to network partitioning are returned to the sender.
2. there is no subsequent network partitioning before all the transactions affected by the previous partitioning have terminated, i.e. there is no multiple network partitioning.
3. network partitioning and site failures never occur concurrently.
4. masters never fail.
5. once the network is partitioned, it will remain in the partitioned state until all the transactions affected by the partitioning have terminated, i.e. there is no transient network partitioning.

Assumption 5 simplifies the problem but will be relaxed later.

5.2. Approach to the problem

Denote the two partitions of the network G_1, G_2 and the boundary between them B (Fig. 4). A site can tell whether partitioning has occurred or not by having a timeout or receiving an undeliverable message. Note, however, that a site does not know which partition it is in and where the boundary is. For a particular transaction, let G_1 be the partition in which the master is located. The basic ideas for solving the problem are as follows :

1. Slaves in G_2 will commit iff there is at least one prepare message flowing through boundary B before partitioning occurs. If there exists such a prepare message, then the master can commit all the slaves in G_1 and the slave (or slaves, since there may be more than one) in G_2 that receives the prepare message will be responsible for committing all the slaves in G_2.
2. If partitioning occurs when the master is in w_1 state waiting for "yes" and if the master times out eventually, then the master can safely abort all the slaves in G_1 since no prepare messages have been generated and none of the slaves in G_2 will commit.
3. If partitioning occurs when the master is in p_1 state waiting for "ack" and if the master times out eventually and there is no undeliverable prepare message returned, then the master can safely commit all the slaves in G_1 since we are sure that all the slaves will receive the prepare message and all the slaves in G_2 will commit.
4. If partitioning occurs when the master is in p_1 state and there is at least one undeliverable prepare message returned to the master, then the master can tell whether there is at least one prepare message flowing through boundary B before partitioning occurs by the following rules :
   a. When a slave times out in p(prepare) state, it will send a probe message: probe(trans_id, slave_id) to the master.
   b. If the probe messages that the master received are sent by exactly those slaves that do not have an undeliverable prepare message returned to the master then there is no prepare message flowing through boundary B and the master can safely abort all the slaves in G_1; else there is at least one prepare message flowing through boundary B and the master can safely commit all the slaves in G_1.
5. If partitioning occurs when the master is in p_1 state and there is at least one undeliverable prepare message returned to the master, then all the slaves in G_1 will time out in p state eventually.
6. A slave which has received a prepare message can tell that it is in G_2 by (1) receiving a returned undeliverable "ack" message, or (2) receiving a returned undeliverable probe message. Such a slave will commit all the slaves in G_2.

5.3. The termination protocol

In this subsection, we describe the termination protocol by specifying the actions to be taken when timeout occurs or an undeliverable message is received in each state.

Notations

$abort_{1-n}$:	{ $abort_1$, ..., $abort_n$ }
$commit_{1-n}$:	{ $commit_1$, ..., $commit_n$ }
UD(msg):	the corresponding undeliverable message of msg.
PB:	the set of slaves from which the master has received probe messages.
UD:	the set of slaves to which the prepare messages sent by the master are undeliverable.
T:	the longest end-to-end network propagation delay.
N:	the set of sites participating in this transaction = { 1, 2, ..., n }.

Master

```
w1 ---  (1) timeout:                send abort1-n

        (2) undeliverable Xact:     send abort1-n

p1 ---  (1) timeout:                send commit1-n

        (2) undeliverable preparei:

            UD := { i } ;
            PB := φ ;
            reset timer 5T ; (Fig. 6)

        xx: wait(event) ;

            case of event {

            receive UD(prepare_j):

                        UD := UD + {j} ;
                        goto xx ;

            receive probe(trans-id, slave_j):

                        PB := PB + {j} ;
                        goto xx ;

            timeout:    if( N - UD = PB)
                        then  send abort1-n
                        else  send commit1-n
            }
```

Slaves (i= 2 to n)

```
wi --- (1) timeout:  reset timer 6T ;(Fig. 7)
                     wait(event) ;
                     case of event {

                     receive a commit:   commit ;

                     receive an abort:   abort ;

                     timeout:            abort
                    }

       (2) undeliverable yesi:
                     send abort1-n ;


pi --- (1) timeout:  send probe(trans-id, slavei) ;
                     wait(event) ;
                     case of event {

                     receive UD(probe):  send commit1-n

                     receive a commit:   commit ;

                     receive an abort:   abort

                     }
```

```
       (2) undeliverable Acki:
                     send commit1-n ;
```

Unfortunately, there is still a fly in the ointment. Consider the following scenario : Let $site_i$ and $site_j$ be two of the sites in G_2. $Site_i$ receives a prepare message, sends out an "ack" message and receives a returned undeliverable ack message. $Site_j$ receives no prepare message and waits in state w. It is possible that while $site_j$ receives a commit sent by $site_i$, $site_j$ is still in state w and has not timed out yet; therefore, $site_j$ will not respond to this commit message. This may cause a serious problem because the commit message could be the only commit message $site_j$ will ever receive. If it misses this message, after it times out in state w and waits for 6T, it will abort the transaction. To deal with this problem, we can add one transition from state w to state c when receiving a commit message to the slaves' three-phase commit protocol (Fig. 8).

5.4. Proof of Correctness

In this subsection, we give a formal proof of correctness of the termination protocol.

FACT1: A slave in G_2 will commit only if
(1) it receives a commit from the master,
(2) it times out in w_i , and then receives a commit,
(3) it times out in p_i , and then receives an UD(probe),
(4) it times out in p_i , and then receives a commit,
(5) it receives an UD(ack_i) in p_i, or
(6) it receives a commit from some slave in G_2 while it is in state w or state p.

FACT2: A site in G_1 will commit only if
(1) the master makes a transition from p_1 to c_1 and sends out all the commits,
(2) the master times out in p_1 and sends out all the commits, or
(3) the master receives an UD($prepare_i$), finds that N - UD ≠ PB and sends out all the commits.

Lemma 4: N - UD ≠ PB occurs only if at least one slave in G_2 receives a prepare message.

Proof: The master will check whether N - UD = PB only if it sends out all the prepare messages and receives an UD(prepare) message. Therefore, all the slaves in G_1 will receive prepare messages.

If none of the slaves in G_2 receives a prepare message, then only those slaves in G_1 receive prepare messages. The slaves in G_1 will time out and send probe messages which will be received by the master. Therefore, N - UD = PB = set of all slaves in G_1.

Q.E.D.

Lemma 5: A slave in G_2 commits iff all slaves in G_2 commit.

Proof: "if" : trivial.

"only if" : Let the slave in G_2 that commits be $site_i$. By FACT1, we know that there are only six possible cases.

(case1): $site_i$ receives a commit from the master. In this case, the master must have received all the "ack"s, made a transition from p_1 to c_1 and sent out all the commits. Therefore, for any $site_j$ in G_2 either it will time out in p_j or commit in c_j. If it times out in p_j, then it will eventually receive the returned UD(probe) message and commit all the sites in G_2.

(case2): $site_i$ times out in w_i, and then receives a commit. Since $site_i$ is still in w_i, the commit it receives must come from some $slave_j$ in G_2. When a slave in G_2 sends out commits (after receiving an UD(probe) or UD(ack)), it will send to all the slaves in G_2. Therefore, all the slaves in G_2 will commit.

(case3): $site_i$ times out in p_i, and then receives an UD(probe). When $site_i$ receives a UD(probe), it will send commits to all slaves in G_2; therefore, all slaves in G_2 will commit.

(case4): $site_i$ times out in p_i, and then receives a commit. Since $site_i$ receives a commit after it times out in p_i, this commit message must come from some slave in G_2. By the same argument as (case2), all slaves in G_2 will commit.

(case5): $site_i$ receives an $UD(ack_i)$ in p_i. When $site_i$ receives an $UD(ack_i)$, it will send commits to all slaves in G_2; therefore, all slaves in G_2 will commit.

(case6): $site_i$ receives a commit from some slave in G_2 while it is in state w or state p. By the same argument as (case2), all slaves in G_2 will commit.

Q.E.D

Lemma 6: A site in G_1 commits iff all sites in G_1 commit.

Proof: "if" : trivial.

"only if" : By FACT2, if a site in G_1 commits, then all sites in G_1 will commit.

Q.E.D.

Lemma 7: A slave in G_2 receives a prepare message iff all slaves in G_2 commit.

Proof: "if" : trivial.

"only if" : Let the slave in G_2 that receives a prepare message be $site_i$. There are three possible cases.

(case1) $site_i$ receives a commit too. By Lemma5, all the slaves in G_2 will commit.

(case2) $site_i$ times out in p_i. $Site_i$ will probe the master and eventually it will receive the returned UD(probe) or a commit sent by some slave in G_2 and commit. By Lemma5, all slaves in G_2 will commit.

(case3) $site_i$ receives an $UD(ack_i)$. $Site_i$ will send out commits to all slaves in G_2; therefore, all slaves in G_2 will commit.

Q.E.D.

Lemma 8: A slave in G_2 receives a prepare message iff all sites in G_1 commit.

Proof: "only if" : Let the slave in G_2 that receives a prepare be $site_i$. Since $site_i$ has received a prepare message, the master must be in p_1 or c_1. If the master is in c_1, then all the sites in G_1 will commit by Lemma6.

If the master is in p_1, there are two cases to be considered.

(case1): The master times out in p_1. The master will commit and send commits to all the slaves; therefore, all sites in G_1 will commit.

(case2): The master receives an UD(prepare) message. Since $site_i$, located in G_2, has received a prepare message and the master can only receive the probes sent by slaves in G_1, the master will find that the set of slaves that probe it does not equal to the set of slaves that have received prepare messages. Therefore, the master will send commits to all slaves in G_1 and all of them will commit.

"if" : By FACT2, there are three cases to be considered.

(case1): The master makes a transition from p_1 to c_1 and sends out all the commits. Since the master will make a transition from p_1 to c_1 only if it has received all the "ack" messages, all the slaves must have received the prepare message.

(case2): The master times out in p_1 and sends out all the commits. Since the master will time out in p_1 only if all the prepare pessages are deliverable, all the slaves must have received the prepare message.

(case3): The master receives an UD(prepare) message, finds that $N - UD \neq PB$ and sends out all the commits. By Lemma4, there exists a slave in G_2 which has received a prepare message.

Q.E.D.

Theorem 9: The termination protocol makes the three-phase commit protocol resilient to optimistic multisite simple network partitioning.

Proof: By Lemma5 to Lemma8, we know that

(1) a slave in G_2 commits implies all the slaves in G_2 commit which implies a slave in G_2 receives a prepare message which means that all sites in G_1 commit.

(2) a site in G_1 commits implies all sites in G_1 commit which implies a slave in G_2 receives a prepare which means that all slaves in G_2 commit.

Therefore, the three-phase commit protocol becomes resilient to optimistic multisite simple network partitioning.

Q.E.D.

In fact, the basic ideas used in designing the above termination protocol can be applied to design termination protocol for any master-slave type commit protocol that satisfies Lemma1 and Lemma2.

Theorem 10: A commit protocol can be made resilient to multisite simple network partitioning if the following conditions are satisfied:

1. there exists no local state that has both a commit and an abort in its concurrency set.
2. there exists no local state that is noncommittable and has a commit in its concurrency set.
3. undeliverable messages are returned to the senders.
4. there is no concurrent network partitioning and site failures.
5. masters never fail.

Proof: For any commit protocol that satisfies conditions (1) and (2), the only adjacent states of a commit state must be committable states and these committable states cannot be adjacent to an abort state. To commit a transaction in such a protocol, there exists a message sent by the master for a slave to make a transition from a noncommittable state to a committable state. Let this message be "m". We can use the ideas listed in Sec. 5.2 to design a termination protocol for this commit protocol in an environment that satisfies conditions (3), (4) and (5) by substituting "m" for "prepare".

Q.E.D.

6. Transient Network Partitioning

A network partitioning is transient if the network recovers before all the transactions affected by the partitioning have terminated. In the previous discussion, we assume that transient network partitioning never occurs in order to simplify the discussion. In this section, we will relax this assumption.

If we take transient network partitioning into consideration, we can enumerate all the possible cases of network partitioning as follows:

(1) no prepare messages pass boundary B.

(2) some prepare messages pass B, some prepare messages do not pass B. This case can be divided into the following two cases :
  (2.1) some ack messages do not pass B.
  (2.2) all the ack messages sent by those sites in G_2 which receive prepare messages pass B. This case can be further divided into the following two cases for transient network partitioning:
    (2.2.1) network recovers at a point in time such that some probe messages do not pass B.
    (2.2.2) network recovers at a point in time such that all the probe messages sent pass B.

(3) all the prepare messages pass B. This case can be divided into the following two cases:
  (3.1) some ack messages do not pass B.
  (3.2) all the ack messages pass B. This case can be further divided into the following two cases:
    (3.2.1) all the commit messages pass B.
    (3.2.2) some commit messages do not pass B. This case can be further divided into the following two cases for transient network partitioning:
      (3.2.2.1) some probe messages sent by those sites in G_2 that receive no commit messages do not pass B.
      (3.2.2.2) all the probe messages sent by those sites in G_2 that receive no commit messages pass B.

Our original termination protocol works well in all cases except case (3.2.2.2). In this particular case, those sites in G_2 that receive no commit messages sent by the master will wait forever. To deal with this problem, we calculate the longest possible time for a slave to receive an UD(probe) message, a commit or an abort after it times out in state p and get the following result:

Case(2.1): T.
Case(2.2.1): 4T.
Case(2.2.2): 5T (Fig. 9).
Case(3.1): T.
Case(3.2.2.1): 4T.
Case(3.2.2.2): ∞ .

Since only case(3.2.2.2) takes more than 5T, a slave can tell that case(3.2.2.2) has happened and commits itself if it has waited for 5T after it times out in state p and receives neither an UD(probe) message nor a commit nor an abort. Therefore, to deal with transient network partitioning, we only have to modify the action to be taken when a slave times out in state p as follows:

```
p_i --- (1) timeout:  reset timer 5T ;
                      send probe( tran-id, slave_i ) ;
                      wait(event) ;
                      case event of {
```

receive UD(probe): send commit$_{1-n}$;

receive a commit: commit ;

receive an abort: abort ;

timeout: commit
}

7. Conclusion

In this paper, we assume that network partitioning and site failures never occur concurrently. This assumption is necessary due to the following two observations: (1) if the only slave in G_2 that receives a prepare message fails before it sends out commit messages, then all slaves in G_2 will abort while all participating sites in G_1 will commit. (2) if none of the slaves in G_2 receives a prepare message and one of the slaves in G_1 fails after receiving a prepare message but before sending a probe message, then all slaves in G_2 will abort while all participating sites in G_1 will commit. In fact, there is no commit protocol resilient to concurrent network partitioning and site failures because site failures in a network partitioning may have the same effect as message loss and it has been proved that there exists no commit protocol resilient to network partitioning when messages are lost [7]. Furthermore, we assume that masters never fail. The reason we make this assumption is that the termination protocol to be taken for network partitioning is different from the termination protocol to be taken for master site failure which has been proposed by Dale Skeen [4] and there is no simple way to distinguish a site failure from a network partitioning which separates that site from the rest of the system. These two assumptions amount to assuming that site failures never occur and a site will abort a transaction only because of network failures, transaction deadlocks with other transactions or the user aborting the transaction. With technological advances of hardware fault tolerance, computers will be very reliable compared to links in the network. Therefore, we feel that assuming site failures never occur is acceptable.

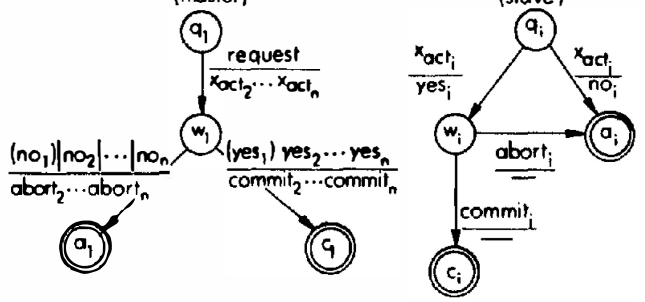


Figure 1: The two-phase commit protocol.

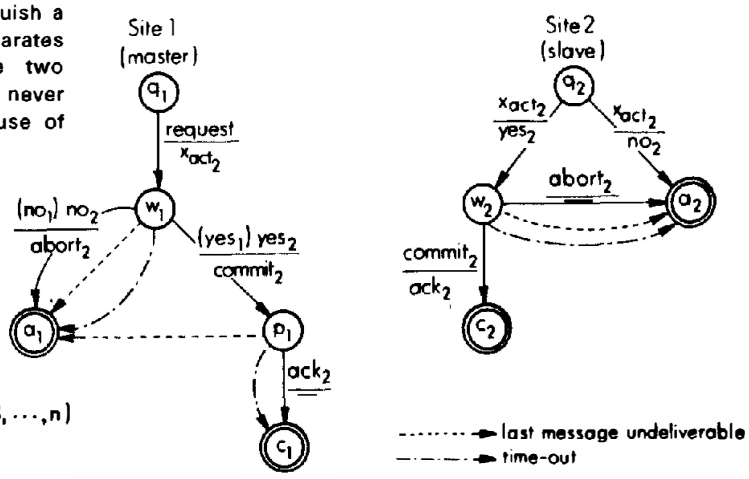


Figure 2: The extended two-phase commit protocol augumented with timeout transitions and undeliverable message transitions (two participating sites).

References

1. J. Gray. Notes on database operating systems. In *Operating System: An Advanced Cource*, New York: Springer-Verlag, 1979.

2. B. Lampson, H. Sturgis. Crash Recovery in a Distributed Storage System. Computer Sci. Lab., Xerox Parc, Palo Alto, CA, 1976.

3. B. G. Lindsay et al. Notes on Distributed Databases. IBM Research, July, 1979.

4. D. Skeen. Nonblocking Commit Protocols. SIGMOD Int. Conf. on Management of Data, 1981, pp. 133-142.

5. D. Skeen. A Quorum-Based Commit Protocol. Proc. 6th Berkeley Workshop, February, 1982, pp. 69-80.

6. D. Skeen. *Crash Recovery in Distributed Database System.* Ph.D. Th., Dept. of EECS, University of California, Berkeley, May 1982.

7. D. Skeen, M. Stonebraker. "A Formal Model of Crash Recovery in a Distributed System." *IEEE Trans. on Software Eng. SE-9*, 3 (May 1983), 219-228.

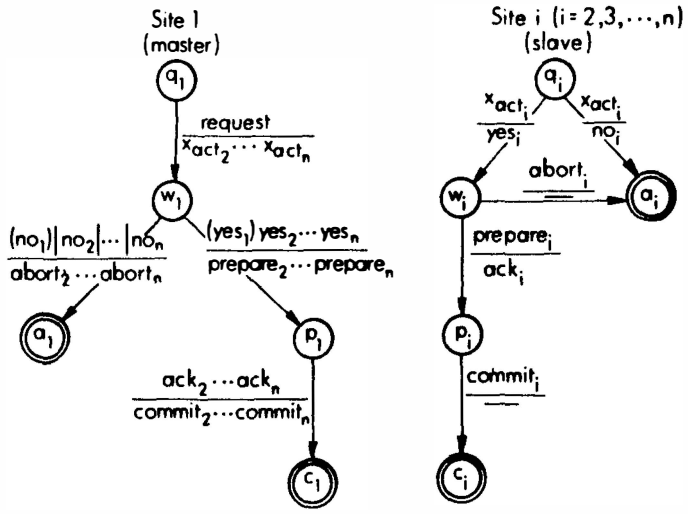
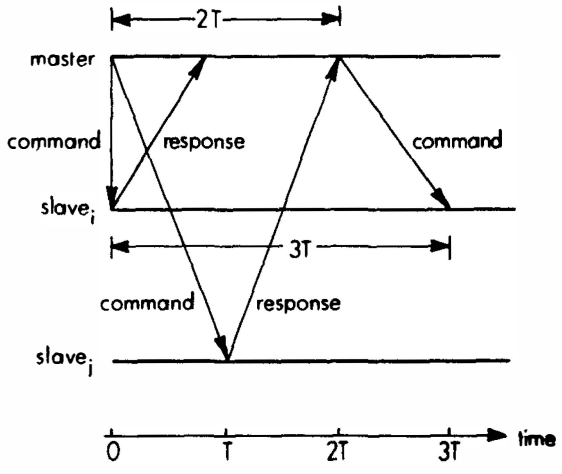


Figure 3: The three-phase commit protocol.



Length of timeout interval for the commit protocol at the master site = 2T.

Length of timeout interval for the commit protocol at slave sites = 3T.

(T: the longest end-to-end network propagation delay)

Figure 5: Timeout analysis.

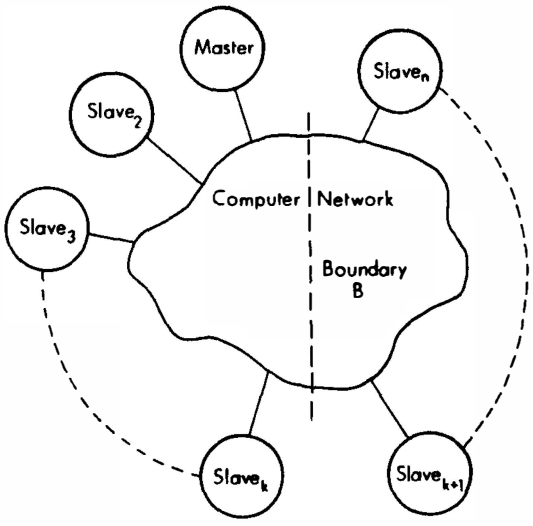
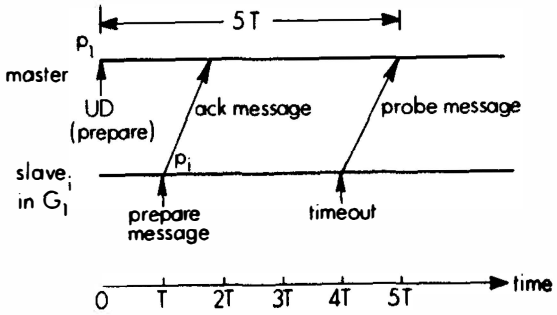
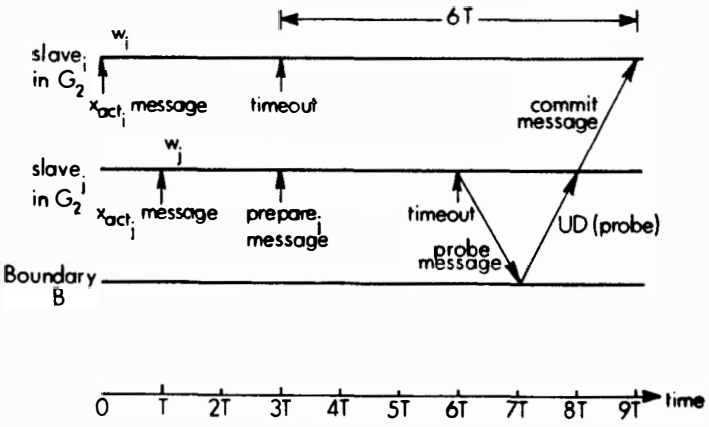


Figure 4: Simple network partitioning.



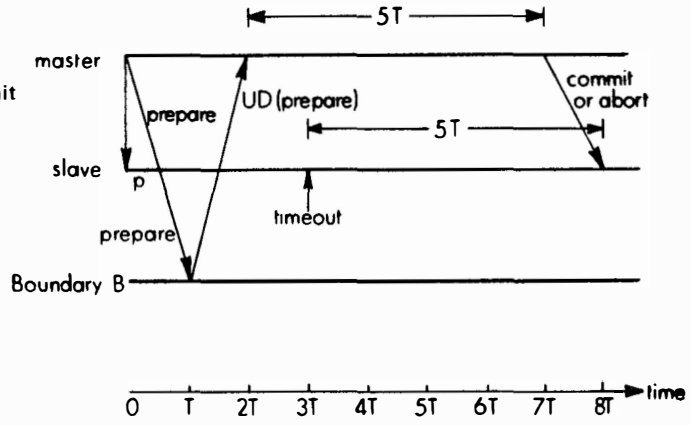
The longest possible time for a master to receive the probe message after receiving an undeliverable prepare message = 5T.

Figure 6: Timing analysis



The longest possible time for a slave to receive a commit after it times out in state w = 6T.

Figure 7: Timing analysis



The longest possible time for a slave to receive an UD(probe) message, a commit or an abort after it times out in state p = 5T.

Figure 9: Timing analysis

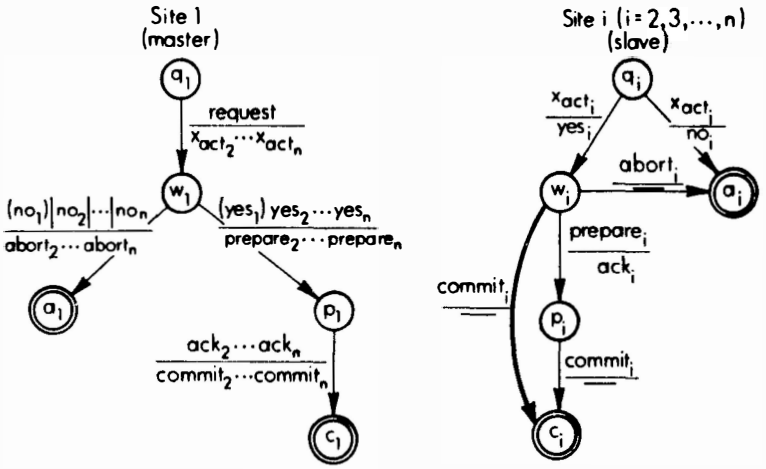


Figure 8: The modified three-phase commit protocol